EECE 478 GROUP PROJECT REPORT

Hyunwoo Choi	34976035
Lance Kersey	22178032
Chris Lee	65659047
Roger Yin	79131041
Adrian Yu	18917039

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

This document describes the development process and final product of a game created to satisfy the 3-dimensional interactive EECE 478 Game Project requirement. The scope of this report primarily includes the graphical and game play features implemented using OpenGL, SDL and C++ in a Windows environment. Also discussed will be features that were considered but abandoned due to time limitations.

The following sections will introduce the story behind the game, a list of the primary features and a user guide outlining the controls. Additionally the rationale, implementation and outcome of each of the features incorporated in the software will be discussed.

Finally, the latter portion of the report will illustrate more technical overview of the software class architecture through low-level class diagrams.

GAME DESCRIPTION

In a galaxy far, far away...

The evil emperor Iverson embarked on an evil journey to rid the universe of promising engineers by unleashing a relentless amount of assignments...

Overwhelmed...the engineers retreated in their spacecraft to a dense asteroid field in the system UBC...

Unyielding...the emperor let loose the imperial guard of automated star fighters to hunt down the rebel engineers...

To combat these menacing threats, the engineers engineered and equipped their spacecraft with advanced weapons and even more advanced shields…

And so the saga continues...

Starwarriors is a single player 3-dimensional space fighter simulator. The objective of the game is to survive the onslaught of Imperial Guard fighters while dodging or destroying asteroids. The main character, the engineer, is given a shielded space fighter equipped with twin ion lasers and an ion canon. Using this space fighter, the engineer (you) is to avoid or destroy all Imperial Guard fighters automated by sophisticated artificial intelligence.

GAME FEATURES

The main game features are divided into the following categories.

Graphical Features

- Space Arena
- View Frustum Culling
- Collision Detection
- 3D Studio Model Loader
- 3D Studio Models
- Texture Mapping
- Particle Engine
- Lighting and Material Variations
- Heads-Up-Display (HUD)
- Radar
- Game Menu
- First Person Camera
- Third Person Camera
- Opening Video

Game Play Features

- Artificial Intelligence
- 3D Space Engine
- Sound Effects
- User Interface
- Save and Load

HOW TO PLAY

Starwarriors has a simple and intuitive game control system, typical of other space fighting computer games, which makes it appealing to both experienced and inexperienced gamers. The movement of the ship is controlled by using mouse and keyboard, and the weapons can be fired by pressing the mouse buttons: left button for the primary weapon and right button for the secondary weapon. This type of control system is very popular and widely used for many conventional flight simulation games currently out in the market. The more detailed keyboard and mouse commands are as listed below.

1. Mouse

- Menu Mode:
  - Mouse movement: Moving the cursor over an item turns the item into red color
  - Left mouse button: Select an item in the menu
  - Right mouse button: Not used
- Game Mode:
  - Mouse movement: Moving up, down, left, or right controls pitch and yaw
  - Left button: Fire the primary weapon (ion laser)
  - Right button Fire the secondary weapon (ion canon)

2. Keyboard

- Game Mode:
  - 'w' key: Accelerate the user fighter
  - 's' key: Decelerate the user fighter
  - 'a' key: Roll the user fighter to the left
  - 'd' key: Roll the user fighter to the right
  - *'Up' arrow key: Pitch the user fighter down
  - *'Down' arrow key: Pitch the user fighter up
  - *'Left' arrow key: Yaw the user fighter to the left
  - *'Right' arrow key: Yaw the user fighter to the right
  - Escape (esc) key: Pause the game and display the instant in-game menu
  - Spacebar key: Fire the primary weapon (ion laser)

*Optional if using a mouse

FEATURE DESCRIPTIONS

Space Arena

Rationale

The Arena class as shown in Figure 28 is responsible for drawing any non-moving (static) objects in the three dimensional game world. This includes the Arena walls, any suns or planets and light sources. This implementation takes complexity away from the engine, allowing the Arena class to manage all static objects so the Engine can focus on dynamic objects.

Implementation

When the user chooses to load a level to play, the Engine class creates an Arena object by passing the length, width and height of the desired movable area to the constructor. Initially the walls were statically programmed to be a set distance away, but later revisions allow the walls to automatically scale themselves such that they are at least ten times farther away from the user than the user can travel. After this scaling factor is determined, the polygon density is calculated such that no matter what size the arena is, the star textures mapped to the walls always appear the same size. Figure below illustrates the wall structure of the arena; initially a cube shaped implementation was used but this resulted in the corners of the cube being much farther away, and thus the stars mapped to those polygons will appear smaller. The new structure is more spherical while still having a low polygon density to relieve this symptom; helper functions are used to draw the one plane and two plane squares in the strcuture, and also the three plane triangles.

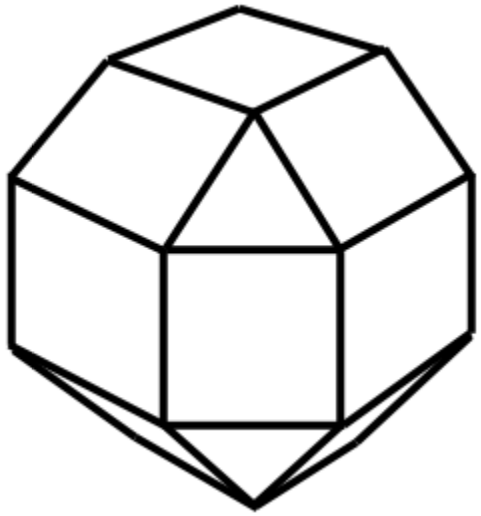

Figure 1 - Arena Wall Structure

Outcome

The final Arena class design performs well and has the flexibility to have an extremely low polygon count when running on a slow system. Future revisions may include support for multiple suns, moons and planets to be positioned, additional light sources and more refined texture artwork.

View Frustum Culling

Rationale

Because Starwarriors has been in development for a matter of months, the arena's polygon count is not extremely high. The fact that it is in outer space also helps, as there are no ground planes or structures to draw. However, the special effects used can be very CPU intensive so by cutting down the polygon count further, higher frames per second can be achieved.

Implementation

View frustum culling offers the ability to skip all code to draw a certain object if it is not in the viewable area of the camera. This differs from built-in culling support which will cull on a vertex-by-vertex basis, therefore still running a lot of unnecessary code. The bounding volume encompassing the viewable area is shown in Figure 2 below; determining which objects reside in this area is achieved in two stages: sphere bound culling and cone bound culling.

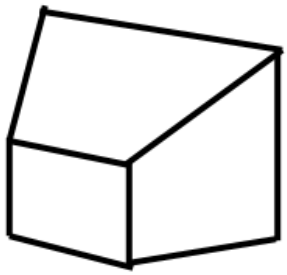

Figure 2 - Bounding Volume of View Frustum

The Sphere culling is relatively efficient and can be used to exclude the drawing of most objects. However, when imagining a sphere that will encompass the view frustum it can easily be seen that a lot of non-viewable area is identified as viewable. The more CPU intensive cone stage is run only when the sphere stage returns true, and offers increased culling accuracy. Cone bounding still does not fully represent the volume, but is close enough. Exact culling can be achieved by representing the view frustum as six planes however, this requires more complex calculations and therefore the trade-off is not necessarily worth it.

Outcome

Starwarriors utilizes both sphere bound and cone bound culling algorithms in a sequential manner which limits the use of CPU, making it an efficient implementation.

Quaternion

Rationale

Quaternions are the most modern technique for tracking the orientation of an object in 3D space. Traditionally, an x, y then z axis angle change would be used to separately define the pitch, yaw and roll changes for each frame. This requires tracking much more data, and also additional code to handle special cases such as gimbal lock (a 90 degree rotation resulting in overlapping axis). Quaternions avoid gimbal lock by applying pitch, yaw, and roll in a manner which is not dependant on the order they are applied, and also requires less data and calculations to do so.

Implementation

The machine representation of a Quaternion is very difficult to understand as it cannot be visualized in 3-dimensional space; thankfully they are simple to use in conjunction with user-translator functions. The primary quaternion functions used in the game are as follows:

- fromEuler(GLfloat yaw, GLfloat pitch, GLfloat roll) [1]
  - Specify the pitch, yaw and roll in the object's local coordinates and update the quaternion with one call. Three quaternions are created, one for each angle update, and these are then multiplied into the existing quaternion. The quaternions are calculated as follows:

```
GLfloat sinp = sin(pitch);
GLfloat siny = sin(yaw);
GLfloat sinr = sin(roll);
GLfloat cosp = cos(pitch);
GLfloat cosy = cos(yaw);
GLfloat cosr = cos(roll);


Quaternion q1(cosp, sinp, 0, 0); // Pitch quaternion
Quaternion q2(cosy, 0, siny, 0); // Yaw quaternion
Quaternion q3(cosr, 0, 0, sinr); // Roll quaternion

multiply(q3, false);
multiply(q2, false);
multiply(q1, false);

normalize(); // Return quaternion to unit size
```

- fromAxis(Vector direction, GLfloat angle) [1]

- Specify the global coordinate system vector to rotate around, and the angle to rotate
- Implementation is similar to a single step in fromEuler, however more calculations are required to use the specified vector.

Outcome

The use of Quaternions reduces the need to store excessive amounts of data and perform trigonometric calculations. As a result, the number of computations and source code is minimized.

Collision Detection

Rationale

Collision detection is extremely CPU intensive. The most basic implementation would check the distance between all objects in the entire arena however, distance calculations use the time costly square root math function which should be avoided whenever possible. For this reason a hierarchical model was chosen which will exclude most distance calculations, offering a significantly increased frame rate.

Implementation

Figure 3 below illustrates the cube bounding volume of the arena. It can be seen that the master (largest) cube is subdivided into eight smaller cubes, which are subdivided further. This occurs up to the maximum recursion depth which is tuned until the greatest frame rate is achieved (experimentally determined to be six). When an object is created or changes position, it registers itself with the PositionTracker class by giving it a position and velocity vector, the hit points worth of damage it does to objects that collide with it, and the radius of its spherical bounding volume. Spherical bounding volumes were chosen for their efficient collision calculations, and therefore the game was designed for their use: asteroids and ship bubble shields are easily represented as spheres. If a new object is being registered it is returned a unique identifier which can be used to identify itself when updating its position.

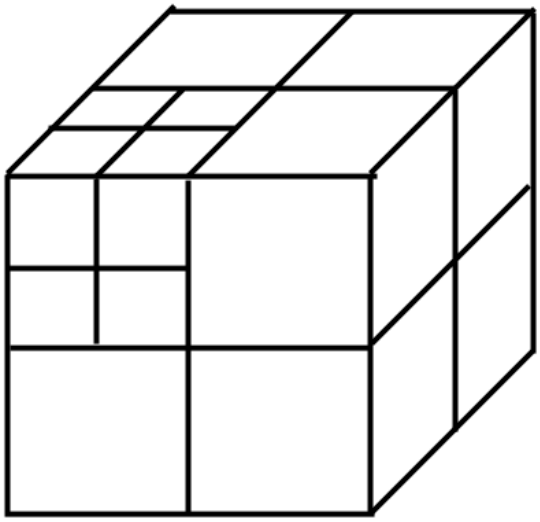

Figure 3 - Hierarchical Culling Structure

The position tracker uses the position and radius of an object to determine the smallest cube in the hierarchical structure that encompasses it entirely, which it will then link it to. Every frame it will recursively analyze the cubes: if an object lies in the same cube as another object, or if one object lies in a parent cube and another object lies in a child cube contained by that parent cube, a distance collision calculation must be performed. It must be recognized that a fast moving object that collides with another object may not be detected if only comparing positions and radii (it will appear to pass right through it), therefore the velocity vector must be used in a time-based calculation. The equation for this calculation when using spherical bounding volumes is as follows:

- t1 = (-2*(ΔP* ΔV) - √(D)) / (2* ΔV)^2
- t2 = (-2*(ΔP* ΔV) + √(D)) / (2* ΔV)^2
  where D = 4*((ΔP* ΔV)^2 - (ΔP - r^2) ΔV^2) [2]

In these equations, t1 and t2 are the times that the spherical bounding volume's borders will intersect; for the purposes of this game we are only interested in t1. ΔP and ΔV represent the change in position and change in velocity between the two objects, respectively. r is simply the sum of the two object's radii. An optimization exists to avoid solving for t1 every time, and that is by first solving D: if D < 0 then there is no solution so no further calculations are required and the square root function is avoided. [2]

While all potentially colliding objects are compared to one another, each collision is flagged in the position tracker. Each frame an object can use its unique identifier to ask the position tracker if it has collided with anything, how many hit points damage was done, and the point of the collision if a visual effect must be drawn.

Outcome

Initially the collision detection was not done in a time sensitive manner however, once it was discovered that the ion laser weapons move too fast for this implementation, the time sensitive calculation was added. Due to the spherical bounding volumes and hierarchical structure of the collision detector, this implementation has proven to perform adequately. Future revisions may add support for different bounding volumes in order to accurately represent more complex shapes.

3D Studio Models

The following sections discuss the 3DS file format loader and the creation of the 3DS models used in the game.

Loader

Rationale

Drawing complicated 3D models using only OpenGL commands is not only time consuming but also very difficult, therefore 3D Studio was chosen to build all the 3D models in Starwarriors. The 3DSLoader is designed to be the bridge between the models created by 3D Studio and the graphics engine. It is capable of importing any models created in 3DS Studio and converting them into formats the graphic engine can interpolate.

Implementation

3DS files are special such that they contain information blocks called CHUNKS. These CHUNKS hold information to describe the model's name, the vertices coordinates, the UV mapping coordinates, the list of polygons, the variation of materials and the key frames of animations. These CHUNKS are not listed in linear format but more like XML format, each consisting of the following fields : CHUNK ID, length of the CHUNK, and CHUNK DATA. A sample CHUNK structure is listed below.

MAIN CHUNK 0x4D4D

3D EDITOR CHUNK 0x3D3D

OBJECT BLOCK 0x4000

TRIANGULAR MESH 0x4100

VERTICES LIST 0x4110

FACES DESCRIPTION 0x4120

FACES MATERIAL 0x4130

MAPPING COORDINATES LIST 0x4140

SMOOTHING GROUP LIST 0x4150

LOCAL COORDINATES SYSTEM 0x4160

LIGHT 0x4600

SPOTLIGHT 0x4610

CAMERA 0x4700

Since the models introduced in Starwarriors are not as complicated, only VERTICES LIST and MAPPING COORDINATES are interpolated. All other CHUNK information is discarded. During loading, the 3DSLoader will load VERTICES LIST and MAPPING COORDINATES into structure type OBJ_TYPE defined in *main.h*, which will be handled by the graphics engine to draw the actual object on to the screen.

Outcome

The 3DSLoader is called in the object initialization process. Although it has limited functionality and compatibility, it works very well with Starwarriors, which contains very few models. Although lacking the ability to load complex objects and animations, it offers little overhead in the form of loading times in return.

Model Creation

Rationale

3D Studio Max (3DS Max) is used to create the models used in the game because it is a gaming industry standard. Additionally, there are an abundant number of online tutorials and references to reduce the learning curve. These models are exported to a 3DS file format which can be interpreted and stored into memory as bytes using C++.

Implementation

Despite the reduced learning curve, creating a 3DS model is still extremely complex. Although creating an object is simple, 3DS Max provides numerous methods to achieve the same result. Exporting and rendering the finished model can be even more complex.

The ship seen in figure 4 below is the model of the User Fighter used in the game. This model is one complete object, transformed and manipulated multiple times from a primitive Box object type in 3DS Max. Originally, the material used for the ship was created within 3DS Max. However, this 3DS material information is excluded from the 3DS file format as it only includes the name of the materials and primitive vertex and texture mapping coordinates.

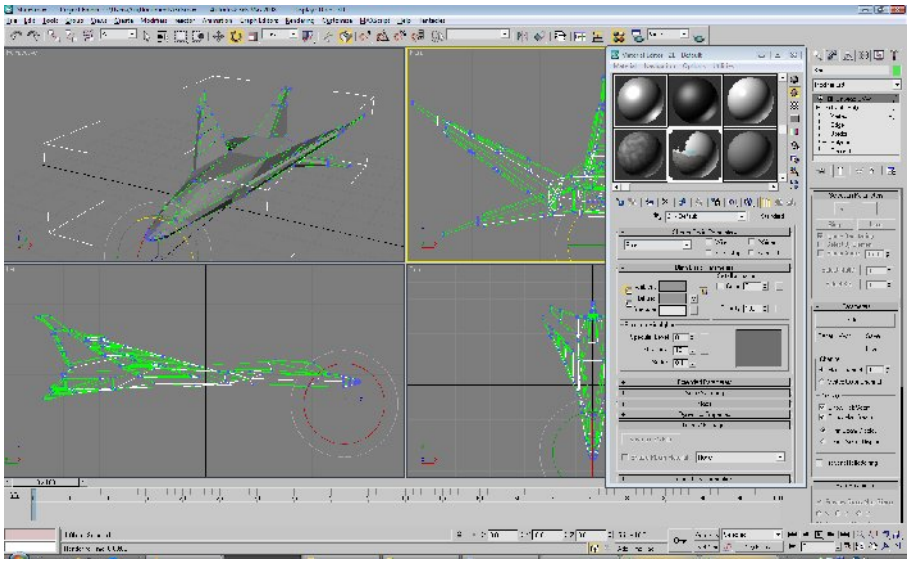

Figure 4 - User Space Fighter Designed in 3D Studio Max

After wasting time trying to debug the defect-free 3DS model loader we created, the problem was traced back to the problem in the process of the material mapping used to create the ship. The corresponding problem was that the original texture map of the ship in 3DS Max was not explicitly texture mapped with an image file. Therefore, the 3DS file exported did not include the proper header bytes used to distinguish an external texture image file. Eventually, proper texture mapping procedures were used and the resulting flattened 2-dimensional texture image (see figures 5 and 6) rendered from 3DS Max was used to texture the in-game models. Due to time constraints, exporting and using animations created from 3DS Max was abandoned.

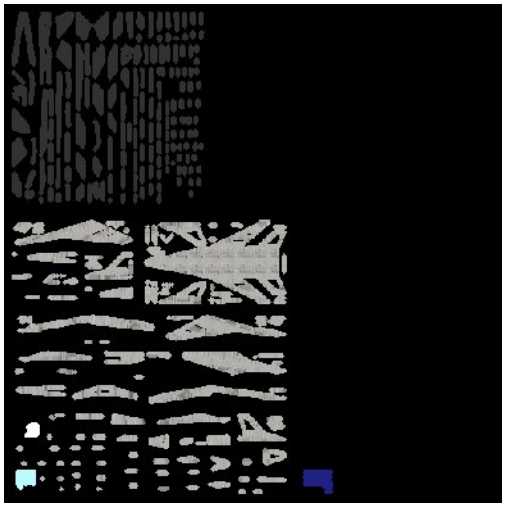

Figure 5 - Flattened 2-Dimensional Texture Image for the User Fighter Model

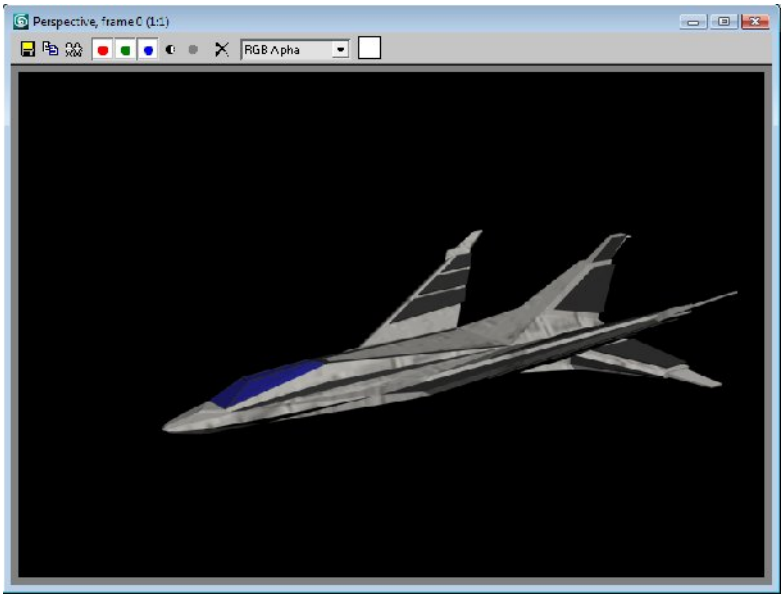

Figure 6 - User Fighter Utilizing Texture Mapping

Outcome

3D Studio Max is a tremendous artistic asset in modeling 3-dimensional objects. Despite its complex feature set and vague help tool, it facilitates the creation of complex 3D models with relative ease in comparison to coding each individual vertex in C++ (User Fighter has 645 Vertices). As a result, all the models used in the game have been loaded from a combination of 3DS and bitmap image files which were created and exported from 3D Studio Max.

Texture/Material Creation and Mapping

Rationale

3D Studio Max excels at both 3-Dimensional modeling and material creation. To further enhance knowledge and background in game design, textures were created using 3D Studio Max rather than using existing ones.

Implementation

Figure 7 below depicts all the materials created and used in 3D Studio Max to give the sun a unique and realistic appearance. To create the materials, a combination of Mix, Noise, and Standard material types are used; a total of 11 different materials are used to create the sun. To enable the object to be exported correctly to a file format recognized by the 3DS file loader, an Automatic flatten UVs modifier is applied to the object. This modifier automatically creates a flattened 2-Dimensional texture map of the Sun and renders it to an external bitmap image file. Following the creation of the bitmap image file, the exported 3DS file of the Sun includes the texture mapping coordinates associated with the rendered

bitmap image file and the material specifications required for OpenGL. All other model materials and texture mapping coordinates are created the same way.

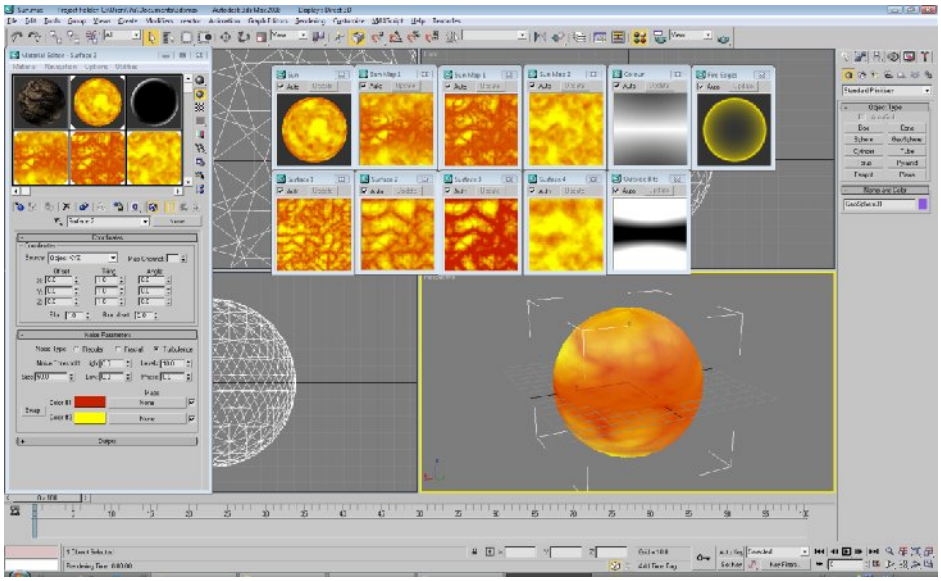

Figure 7 - Materials Used to Texture the Sun

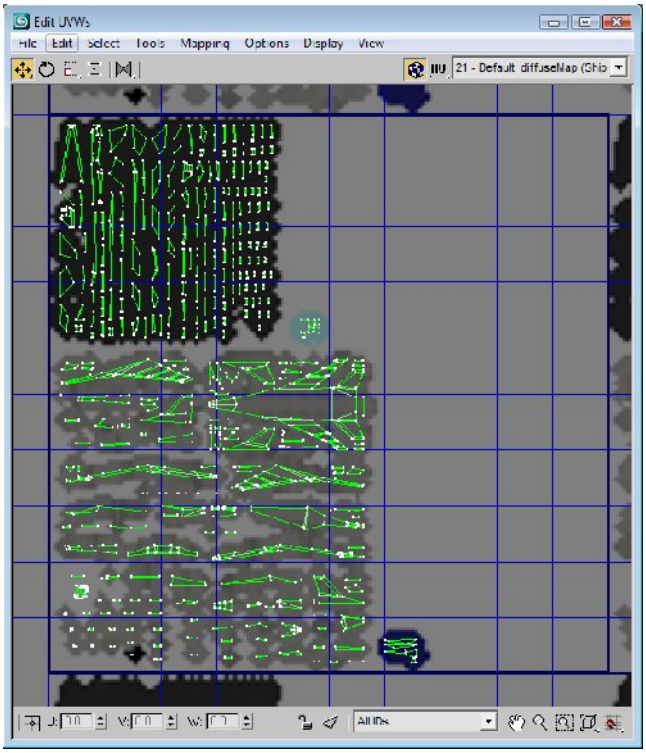

Figure 8 - Texture Mapping Coordinates for the User Fighter (Outlined in Green)

Outcome

All the models used textures created and rendered from 3D Studio Max. Additionally, all the 3DS object files created and exported from 3D Studio Max have the vertices list, the texture mapping coordinates, the materials and the names of the image files used to texture map the objects in OpenGL. As a result, all the models appear more realistic in the game.

Particle Engine

Rationale

A particle engine is essential for creating any special effects such as fire, water or an explosion. In essence it creates the bells and whistles for the game. In short, a particle engine is a driver for a group of individual particles, or points, with different parameters in color, light intensity, velocity, life, and size.

Implementation

In Starwarriors, the particle engine is encapsulated in the Particles class. Each instance of Particles contain an array of at most 1000 points with a list of variables. A sample structure is shown below.

```
typedef struct                          // Create A Structure For Particle
{
        bool    active;                 // Active (Yes/No)
        float   life;                   // Particle Life
        float   fade;                   // Fade Speed
        float   r;                      // Red Value
        float   g;                      // Green Value
        float   b;                      // Blue Value
        float   x;                      // X Position
        float   y;                      // Y Position
        float   z;                      // Z Position
        float   xi;                     // X Direction
        float   yi;                     // Y Direction
        float   zi;                     // Z Direction
        float   xg;                     // X Gravity
        float   yg;                     // Y Gravity
        float   zg;                     // Z Gravity
}
particles;
```

During instantiation, the constructor will give all the points in the particle array random values for Position and Gravity. This by default will create an effect similar to an explosion. Changing gravity to a more uniformed value will give all the particles in the array the same direction of travelling, which in

turn creates the effect of an engine blast. In order to achieve higher quality, a texture is loaded to make the particles more realistic.

Outcome

The use of the particle engine greatly increased the CPU's burden for rendering, as each particle class requires the CPU to perform thousands of calculations per frame. In order to speed things up, less points are created per particle at the expense of visual detail.

Artificial Intelligence

Rationale

Starwarriors is a single player 3-D space fighter simulator. Due to the single player aspect of the game, artificial intelligence (AI) is a necessary feature in the game. The AIFighter class is used to control the movements of and draw the AI Fighter, enhancing the entertainment value and difficulty of the game.

Implementation

The major goals of the AI fighters are to track the user fighter and fire the laser weapon when in range. The major steps of implementation of artificial intelligence are shown in Figure 9.

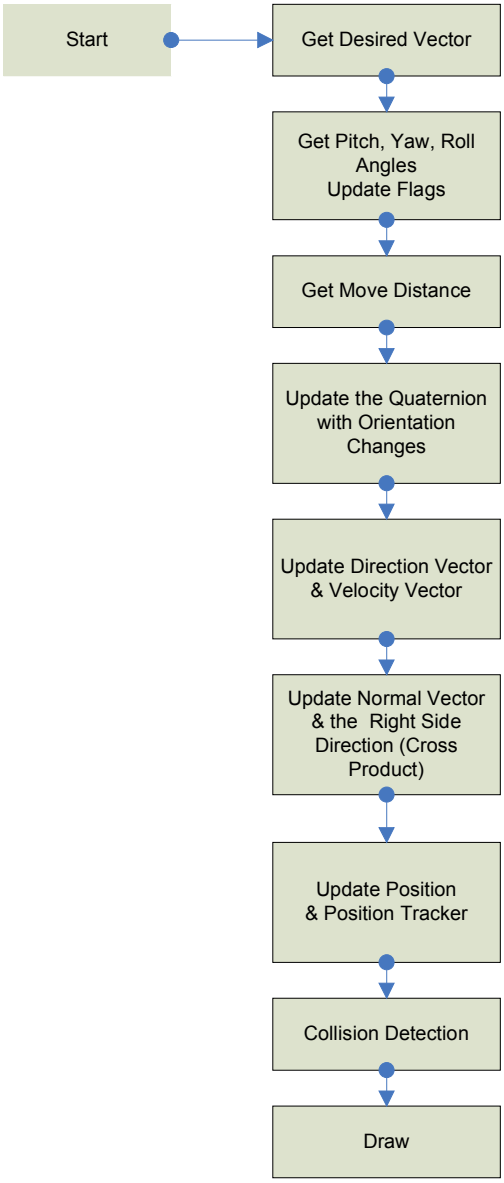

Figure 9 - Artificial Intelligence Flow Chart

The most challenging part of the AI implementation is tracking the user fighter since the orientation of the artificial intelligence model must be considered when updating the position of the AI model. The first step is to obtain desired pitch and yaw angles; when the they are obtained, it is necessary to limit actual angle that the AI will pitch or yaw at the current frame by comparing to the maximum allowed angle for that frame. The move distance is then scaled based on the time passed since the last frame.

In order to track pitching and yawing, a Quaternion is used. The Quaternion is updated every frame based on the calculated pitch and yaw angles. Using the updated quaternion, orientation matrix can

easily be derived. The new position of the model is obtained by adding the time scaled velocity vector to the last known position of the model.

Weapons fire control is relatively simplistic. When the current direction vector is equal to the desired vector, it is known that the AI fighter is facing the user fighter. When this occurs, within a tolerance, a firing flag is set resulting in repeated fire until the firing flag is cleared.

Mathematics for Obtaining Pitch and Yaw Corrections

In order to track the user fighter, the AI should calculate the vector connecting the AI fighter and user fighter. Initially the approach was to calculate a new velocity vector that AI fighter will obtain at next frame using current velocity vector and desired direction vector. By adding the current velocity vector and desired direction vector with scaling, AI is able to obtain the new vector which will be the velocity of AI fighter at next frame. This approach was successful at tracking the user fighter in 3D space however, this method has a crucial problem: It is not designed with orientation in mind, and unfortunately adding support for this proved to be more difficult than redesigning the implementation.

In order to solve this problem it was decided to update the AI fighter based on orientation changes, causing the velocity vector to be trivial to derive. Once the orientation of the AI fighter is obtained, the AI fighter can move forward with a maximum acceleration and velocity. In order to update the orientation, it is necessary to know the AI fighters pitch, yaw, and roll correction angles in terms of the AI fighter's local coordinates, rather than global arena coordinates.

For simpler calculations, only pitch and yaw correction angles are corrected while no roll is specified. However, there would be no significant difference without roll angle.

Figure 10 illustrates how the pitch and yaw can be obtained by specifying a desired vector. α represents the yaw angle while β is the pitch angle, and the angle between the normal and temporary vectors should be 90 degrees at all times.

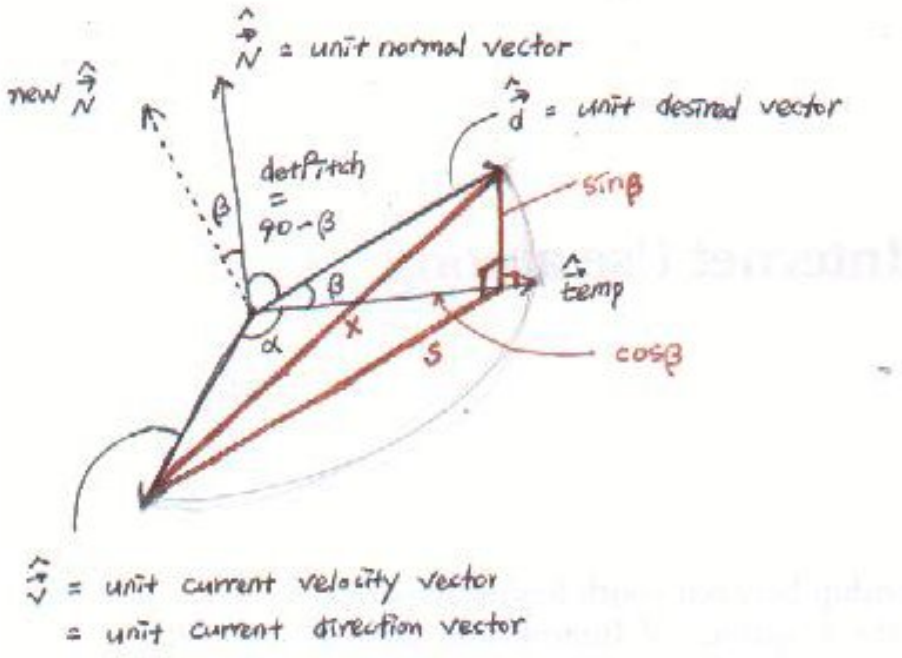

Figure 10 - Vector Representations

The pitch angle is simple to calculate. Since the angle between the normal vector and temporary vector is 90 degrees, the pitch angle is (90° - detPitch), where detPitch is the angle between the normal vector and desired direction vector.

detPitch = acos(unit normal vector ▪ unit desired direction vector) [4]
Note: ▪ represents dot product

pitch angle = $\beta = 90^\circ$ - detPitch

Next the yaw angle can be obtained which is a more complex calculation. Using the brown right-angled triangle in Figure 10 (re-illustrated in Figure 11), the following equations are obtained.

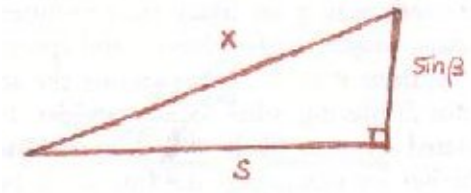

Figure 11 – Triangle used to calculate

$$X^2 = S^2 + \sin^2 \beta$$

$$S^2 = X^2 - \sin^2 \beta \qquad \text{Eq. 1}$$

Since the unit current velocity vector and unit desired direction vector are of length one, it is possible to calculate the value of X when the angle Θ is obtained.

Θ = acos(unit current velocity vector ▪ unit desired direction vector) [4]

$$X = \sqrt{(\| unit\_current\_velocity\_vector \|^2 + \| unit\_desired\_vector \|^2 - 2\cos\theta}$$
$$= \sqrt{2 - 2\cos\theta} \qquad \text{Eq. 2}$$

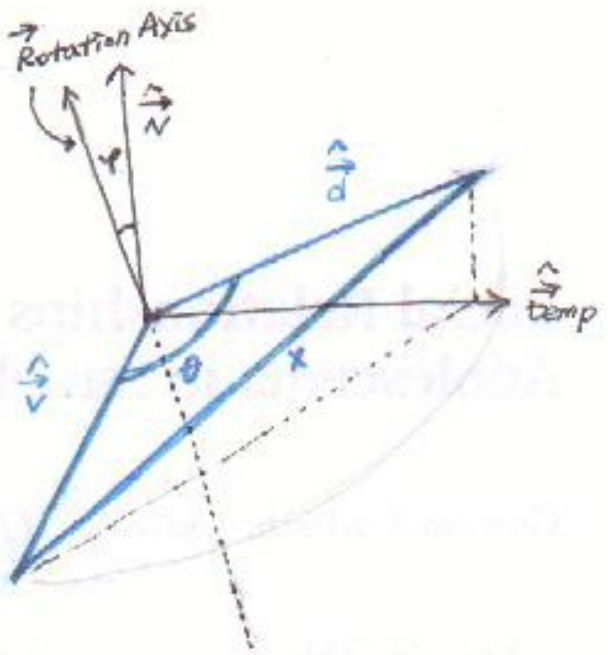

Figure 12 – AI Triangle 1

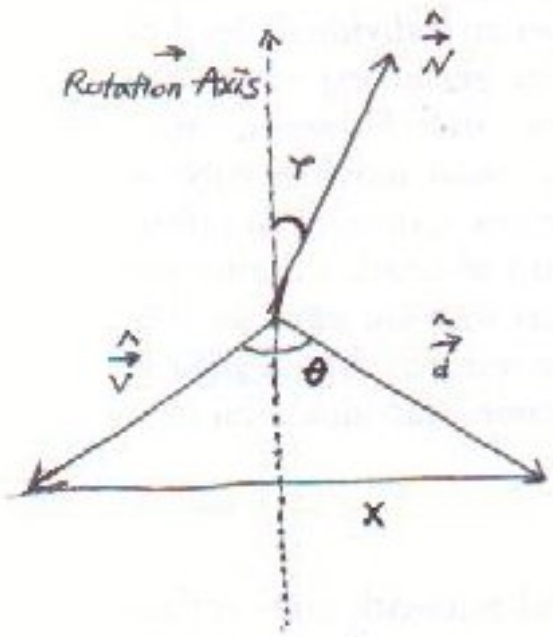

Figure 13 – AI Triangle 2

One more equation is needed to calculate S. Using the left triangle in Figure 14, another equation is obtained.

$$S^2 = \| unit\_current\_velocity\_vector \|^2 + \cos^2\beta - 2\cos\beta\cos\alpha$$
$$= 1 + \cos^2\beta - 2\cos\alpha\cos\beta$$

Eq. 3

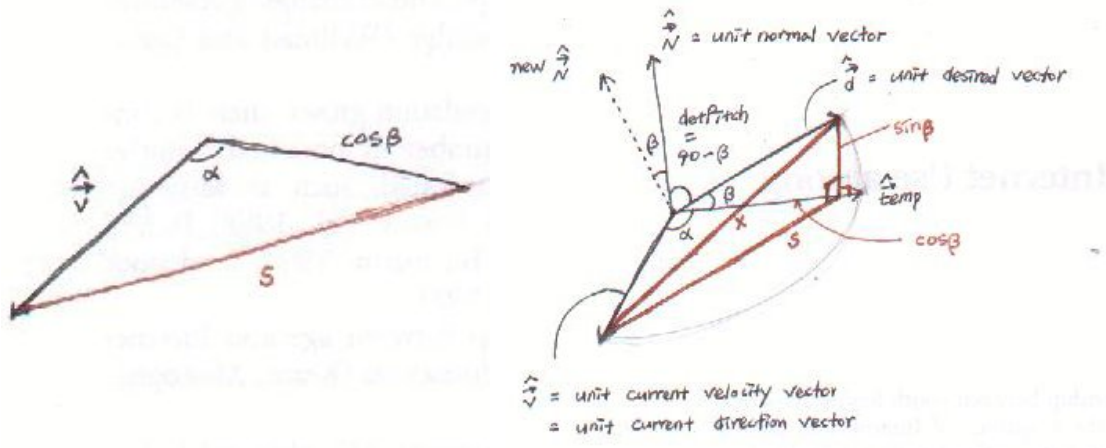

Figure 14 – AI Triangle 3

Using equation 1 and 2, we are able to get the following equations.

$$S^2 = 1 + \cos^2\beta - 2\cos\beta\cos\alpha = X^2 - \sin^2\beta$$
$$1 + \cos^2\beta + \sin^2\beta - X^2 = 2\cos\alpha$$
$$2 - X^2 = 2\cos\beta\cos\alpha$$
$$\cos\alpha = \frac{2 - X^2}{2\cos\beta}$$

(Note: X is from equation 2, and β is the pitch angle.)

From the last equation above, α can be calculated. However, the angle here is the absolute value of the actual yaw angle. The absolute value of the yaw angle must be multiplied by yaw flag in order to get the actual value of yaw angle.

In order to set the yaw flag, we need to get the angle τ in Figure 12 and 12.

τ = acos(unit Rotation Axis vector ▪ unit desired direction vector) [4]

(Note: ▪ represents dot product)

If τ is greater than 90°, the yaw flag is set to -1, else if τ is less than 90°, the yaw flag is set to 1. If the yaw angle zero, then yaw flag is set to zero.

Outcome

The current AI is a fairly simple design. However, the complex math and quaternion used can be applied easily to future extensions or modifications to the AI implementation. It can even be argued that the current AI is too difficult to play against as it has no artificial calculation errors.

Ion Weaponry

Rationale

Ion weaponry is a term used to describe a laser-like weapon in the Starwarriors videogame. They are a simplified version of the user fighter, accepting only a desired direction and location which will continue to be travelled until colliding with another object, or a specified timeout.

Implementation

Once the user fighter class was programmed, the ion weaponry was simplistic to program as all the necessary code was finished. A parent class, IonGun, contains functionality common to both the IonLaser and IonCanon child classes. The child classes simply specify their own maximum velocities and draw functions. At the instance a weapon is fired, space for a new IonGun object is allocated to the end of a linked list. This link list contains all weapons fired from the user fighter that have not timed out. The IonGun constructor requires the position, direction vector, velocity vector and orientation quaternion of the user fighter at that instance. With that information the class can easily update the position every frame

Outcome

The linked list implementation proved to be highly efficient, allowing the weapons to be animated until they are only a pixel in size on the screen without encountering memory overflows. The linked list also allows a fast means to remove an object from the list without the need for array shifting or other statically allocated memory techniques.

Asteroids

Rationale

Asteroids operate very similar to the ion weaponry with one additional feature. The ability to specify two desired rotation angles which should be applied every frame. This means the asteroid can be initialized with a seemingly random velocity and rotation, to offer more realistic arena interaction.

Implementation

The code is near identical to that of the ion weaponry, except the constructor takes in two additional arguments (pitch and yaw angles to be applied every frame). The Engine class manages the asteroids using a linked list, similar to the ion weaponry linked list in the user fighter class.

Outcome

The asteroid fields appear to be very realistic due to the texturing and seemingly random "floating". However, there was insufficient time to support bounce collisions, therefore when two asteroids collide one of them is destroyed.

Graphical User Interface (Heads Up Display)

The following sections discuss the rationale, implementation and outcome of the components that form the Graphical User Interface, otherwise known as a Heads Up Display or HUD.

Rationale

A graphical user interface is an essential component in conveying the game statistics to the user. To facilitate the user input, SDL was chosen because of the popularity among online game development forums.

Implementation

User Interface

As mentioned in "How to Play" section, our program allows user to select any item in the menus and to control the user fighter with a keyboard and a mouse. This can be done because our game is designed as an event driven program. In SDL, whenever an event happens, it is put on the event queue. The event queue holds the event data for every event that happens. For example, if the user were to press a mouse button, move the mouse around, then press a keyboard key, the event queue would be like as shown in the figure below.

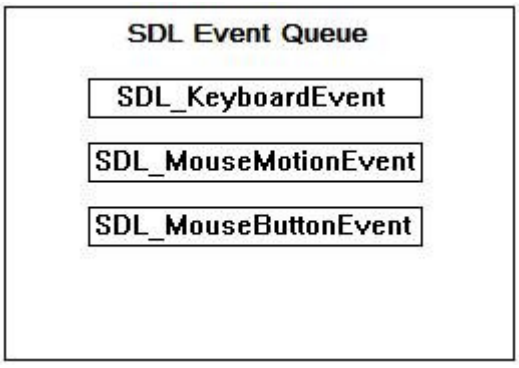

Figure 15 - Putting Event Data on the Event Queue

We then use “SDL_PollEvent()” function to take an event from the queue and sticks its data in the event structure. Figure # shows what “SDL_PollEvent()” function does.

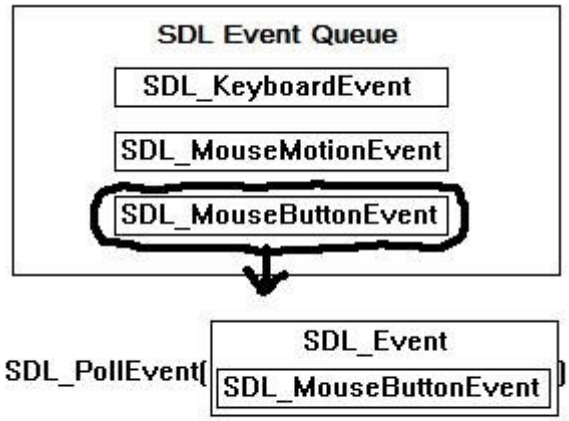

Figure 16 - Polling an Event from the Event Queue

In our game engine, “SDL_PollEvent()” is used with “while” loop in order to keep getting event data while there are events remained on the queue.

All of the keyboard and mouse input functions are written in the “Input” class, and “userFighter” class and “Sound” class are connected to the “Input” class with using pointers. With this approach, control the movement and firing of the user fighter with a desired input, and also we can play the appropriate sound whenever user presses a key or a mouse button. Our program checks for event type “SDL_KEYDOWN” or “SDL_MOUSEBUTTONDOWN” to detect the press of a key or a mouse button. Similarly, the release of a key or a mouse button can be recognized by checking for event type “SDL_KEYUP” or “SDL_MOUSEBUTTONUP.” Mouse movement can be detected by checking the even type “SDL_MOUSEMOTION.” The functions responsible for processing the keyboard and the mouse input are called in the “HandleInput()” function, which is a member function of the main engine.

Game Metrics

Timer

Our program has “Timer” class with “start(),” “stop(),” “pause(),” and “unpause()” functions. Since SDL provides a “SDL_GetTicks()” function, which allows us to get the current clock time, we can easily control the timing by setting up some variables and flags such as “startTicks,” “pausedTicks,” “started, “ and “paused.” For example, to build the “start()” function, we can simply set the “started” flag true and “paused” flag false, then write “startTicks = SDL_GetTicks();” to obtain the current clock time from the beginning. For the “pause()” function, when the “started” flag is true and the “paused” flag is false, the

program changes the “paused” flag to true and execute a line “pausedTicks = SDL_GetTicks() – startTicks; ” to calculate the paused ticks. Other functions are built with using similar techniques: flag changes and time calculating with “SDL_GetTicks()” function. For our game project, the member functions of this “Timer” class are used to cap the frame rate, to calculate the play time, and to handle any kind of time driven things.

Status Display

The status of the game play and the user fighter is shown on the screen all the time while in the game mode. It currently displays the play time (timer), frame counter (FPS), and the speed (thrust) of the user fighter. All the functions are included in the “Status” class, and this class can be added and modified for the demo version; for example, user fighter’s hit point, enemy fighter’s hit point, and some other message relating to the mission objective can be added. With using pointers, we can connect any class to the “Status” class to get the values or strings that we wish to display on the screen while playing.

Radar

The radar is intended to indicate the location of asteroids and enemy artificial intelligence fighters, from the perspective of the user fighter. To achieve this, the user fighter must be displayed in the centre of the radar with the front of the ship facing the top of the screen at all times. Anything above the user icon is dead ahead, everything below is behind, and off to one side represents to the left or right.

To implement this, a separate view frustum is displayed in the lower right corner of the screen, on top of the main view frustum as shown in Figure 17 below. It is also configured such that this second frustum is transparent except for objects drawn in it, as to minimally obscure what is underneath in the main view frustum. The camera is positioned such that it is directly above the user fighter at all times with the normal of the camera equal to the unit direction vector of the user fighter. After positioning the camera, the user fighter, asteroids and enemy artificial intelligence fighters are drawn in their global coordinates as small, low-resolution spheres; orientation is thus not necessary. Colors distinguish the different types of objects, and the spheres will appear smaller (farther away from the bird’s eye camera) if the user is to pitch down, and larger if they should pitch up.

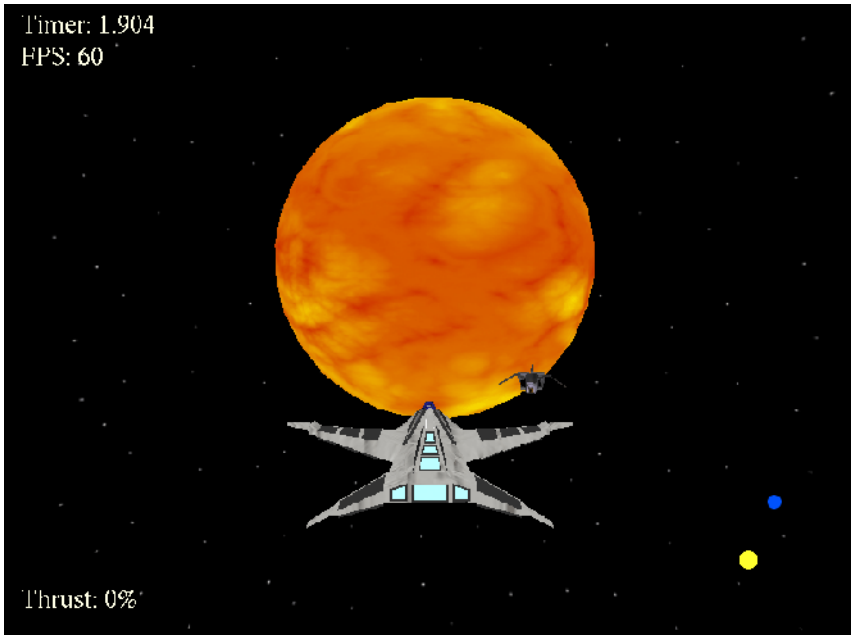

Figure 17 - Illustration of Radar Positioning

Outcome

This implementation reduces flexibility in areas such as displaying out-of-range objects on the very edge of the frustum; however it completely removes the necessity for additional quaternion and vector calculations.

Game Menu

Our game program has two menus, start menu and in-game menu, and all of the menu items are drawn by using OpenGL's bitmap font. Any function related to the menu rendering and menu item processing is in written in the "Menu" class. We have written four different kinds of bitmap font rendering functions in the "Font" class, which is connected to the "Menu" class with a pointer. The "Font" class contains four member functions that allow us to render bitmap strings, spaced bitmap strings, vertical bitmap strings, and stroke font strings. Our program also has the save and load feature, which are defined in the "Save_Load" class.

Rationale

Game menus are an essential component of the game which is used to query the user for their intentions every time the start the game and when they are playing it.

Implementation

Start Menu

Start menu should appear as soon as the game; thus, we use the loop that handles the menu rendering and menu item processing prior to the main loop responsible for the game mode. The “LoadBMP()” function from the “texture” class is used for the 2D texturing in order to display the background picture of the menu, and the “play_main_bgm()” function from the “Sound” class is called for playing the background music. Figure 18 shows the start menu of our game.

Figure 18 - Start Menu

As shown in the Figure 18, there are three items in the start menu: “Start,” “Load Game,” and “Exit.” Each item will turn into red whenever the mouse cursor is placed over. This mouse-over feature can be achieved by monitoring the movement of the mouse; the bitmap font rendering is updated with the new color setting according to the mouse position. Figure 19 shows what happens when the user places the mouse cursor over the “Start” item.

Figure 19 - Start Menu with 'Start' selected

Item selecting feature can be achieved by monitoring the mouse motion and button event; the program can process each item's function whenever the mouse button is pressed in the specific range of the x and y coordinates, where each item is located. The user can enter the game mode immediately, by selecting "Start" with a mouse click. "Load Game" enables the user to begin the game with the latest saved data. Save feature can be implemented by outputting a file with important game information written on it: the user fighter's and enemy fighter's position, speed, and hit point. Load feature can be realized by reading in the data from the saved file and rendering the whole thing according to the information obtained from the file. "Exit" item simply just terminates the game.

In-Game Menu

In-game menu is a quick menu that appears whenever the game is paused. It is just a collection of bitmap font renderings, and there are 5 items: "Resume," "Mission Objective," "Save Game," "Load Game," "Exit." The approaches used for its mouse-over feature and item selecting feature are the same as the ones used for the start menu. The functionality of each menu item is really obvious and intuitive, so they do not need explanation. Unlike the start menu, in-game menu is not using its on screen to display, and this can be possible by not clearing the color buffer and depth buffer. Figure 20 shows the in-game menu.

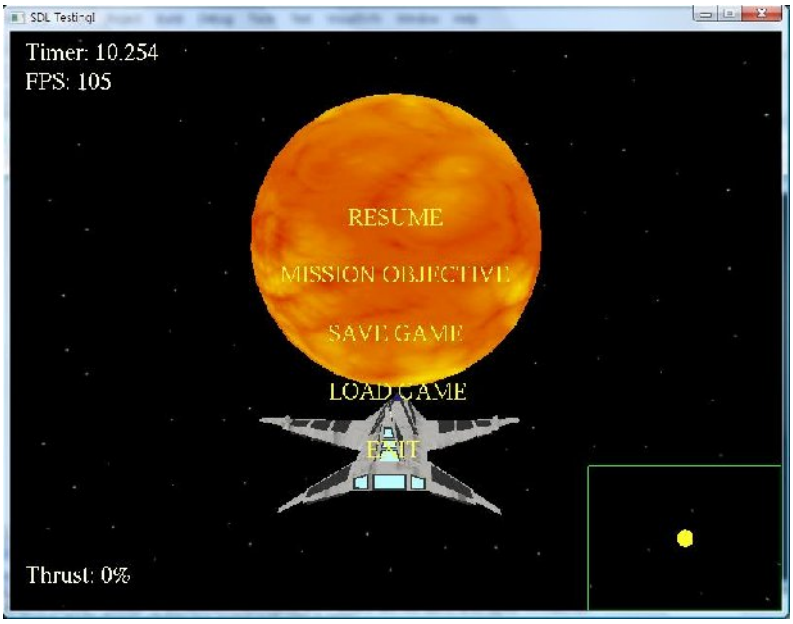

Figure 20 - In-Game Menu

Save/Load Feature

As mentioned earlier, the "Menu" class has a connection with the "Save_Load" class in order to perform the outputting/inputting the game data. Save feature is achieved by outputting a file with using "fstream." Once the "SaveGame()" function is called, it generates a file called "game_save.sav" and then the user fighter's and enemy fighter's position, speed, and hit point are stored in the float value format. It also stores the stage information. Our program calls "LoadGame" function from the "Save_Load" class whenever the "Load Game" is selected. As soon as the function is called the program starts to read in the data from the "game_save.sav" file and stores each data to its own variable. Based on the information stored in the variables, all the objects and arena are rendered, so the user can begin playing the game from the point when he/she saved.

Outcome

There are 2 menus used to facilitate the user's desired game mode. The menu classes are designed to be modular which enables additional features to be seamlessly integrated in the future.

Sound

Rationale

Although adding sound feature to our game project is not one of the basic requirements, playing sound is another key concept of game programming. SDL has some native sound functions; however, an extension library called SDL_mixer is used, since SDL's native sound functions can be quite confusing.

Implementation

All the functions and variables related to the sound feature are included in “Sound” class. This “Sound” class is connected to the main engine as a child class. As soon as the main engine makes a new object of the “Sound” class, two member functions are called, “init_sound()” and “load_file().” The “init_sound()” function initializes SDL_mixer, and the “load_file()” function loads all of the wave files and mp3 files used in our game and stores them into appropriate data typed pointers; Mix_Music is the data type for background music and Mix_Chunk is the data type for sound effects. Once this process is done, our program can play or stop sound by calling functions such as “play_main_bgm(),” “play_weapon(),” “stop_main_bgm(),” and so on.

Our “Sound” class also has a function called “clean_up(),” which contains “Mix_FreeChunk()” to get rid of the sound effects and “Mix_FreeMusic()” to free the music. It also calls “Mix_CloasAudio()” at the end of the function in order to terminate the SDL_mixer. Our main engine calls this “clean_up()” function when terminating the game.

Outcome

Starwarriors utilizes the SDL_mixer library extension to play sound effects. The addition of sound to the game extends the game play value of the product.

Camera

The following sections describe the first and third person views and how the camera orientation is calculated.

Rationale

The camera and its subsequent views are a vital part of any game. The camera’s perspective enables the user to interpret that he is actually moving in the game. Starwarriors incorporates 2 different views to simulate the movement of the star fighter.

Implementation

First Person

A first person camera is very simple to achieve. With the gluLookAt() function, the camera position point, the point it is looking at, and the camera normal vector must be specified. The position is that of the user’s fighter, while the position to look at can be obtained by adding a unit length representation of the velocity vector to the position point. The velocity and normal vectors are obtained by specifying these vectors in the user fighter’s local axis, then multiplying that vector by the quaternion.

Third Person

Third person is a continuation of the first person mathematics. The gluLookAt function is still used however, the position point is specified by applying a camera quaternion to the user fighter's unit direction vector; the cameral quaternion is identical to the user fighter's quaternion except an additional pitch is multiplied in to the direction and normal vectors such that the camera looks slightly down (see "Side View" in

Figure 21); subtracting this vector from the user fighter's position point results in the camera's position. Additionally, in order to partially show the side of the user fighter during a pitch or yaw, the camera direction vector is also adjusted depending on the percentage of pitching and yawing that is occurring (see the effect of yawing in "Top View" of

Figure 21). The current percentage thrust of the user fighter will also scale the z-coordinate of the camera's position in order to give the appearance of accelerating or decelerating.

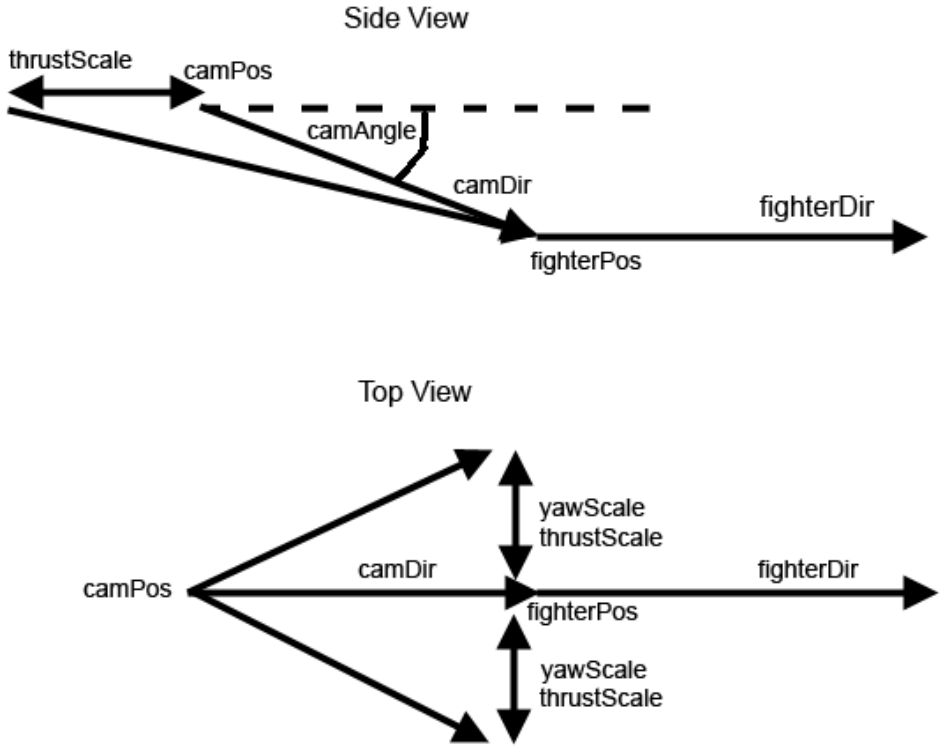

Figure 21 - Third Person Camera Positioning

Next the point to look at must be specified. In the most basic case this is the user fighter's position point, however in order to further make the fighter appear to accelerate or decelerate, a weighting factor dependant on the percentage thrust is multiplied in to the x and y coordinates; this causes the fighter to move away from the centre of view during a turn if the thrust is greater than zero, offering a visual cue that there is a turn with a forward movement occurring. The x and y coordinates are also slightly scaled with the yaw and pitch percentages to offer an even greater visual cue of turning.

Outcome

Overall the camera positioning offers a realistic experience of the user fighter's movement. Due to the advanced nature of this positioning, many vector and quaternion multiplications must be made; this is something which could be optimized in future releases.

Opening Video

Figure 22 - Screenshot of Adobe Premier

Rationale

The purpose of the opening video is to enhance the game by providing a visually appealing story line. Adobe Premier was chosen as a video editor because of its relative popularity among online forums. The parallel use of 3D Studio Max as a material, texture and model creator made it a simple choice to create realistic game play videos using the exact same models used in the game.

Implementation

Adobe Premier is used to compile the video with clips rendered from 3D Studio Max using the exact same models used in the game. Utilizing the opening theme to Star Wars, we are able to recreate a similar opening video utilizing the Title function in Adobe Premier. Additional video effects such as

Transform and Basic 3D are used to simulate the Star Wars opening theme. The game play video is created using keyframe animation in 3D Studio Max.

Outcome

Inspired by multi-million dollar games, the opening video is intended to stimulate the user's interest in the game. The video also provides an effect means of conveying the story of the game.

SOFTWARE ARCHITECTURE

Game Code Flow Diagrams

Global Structures

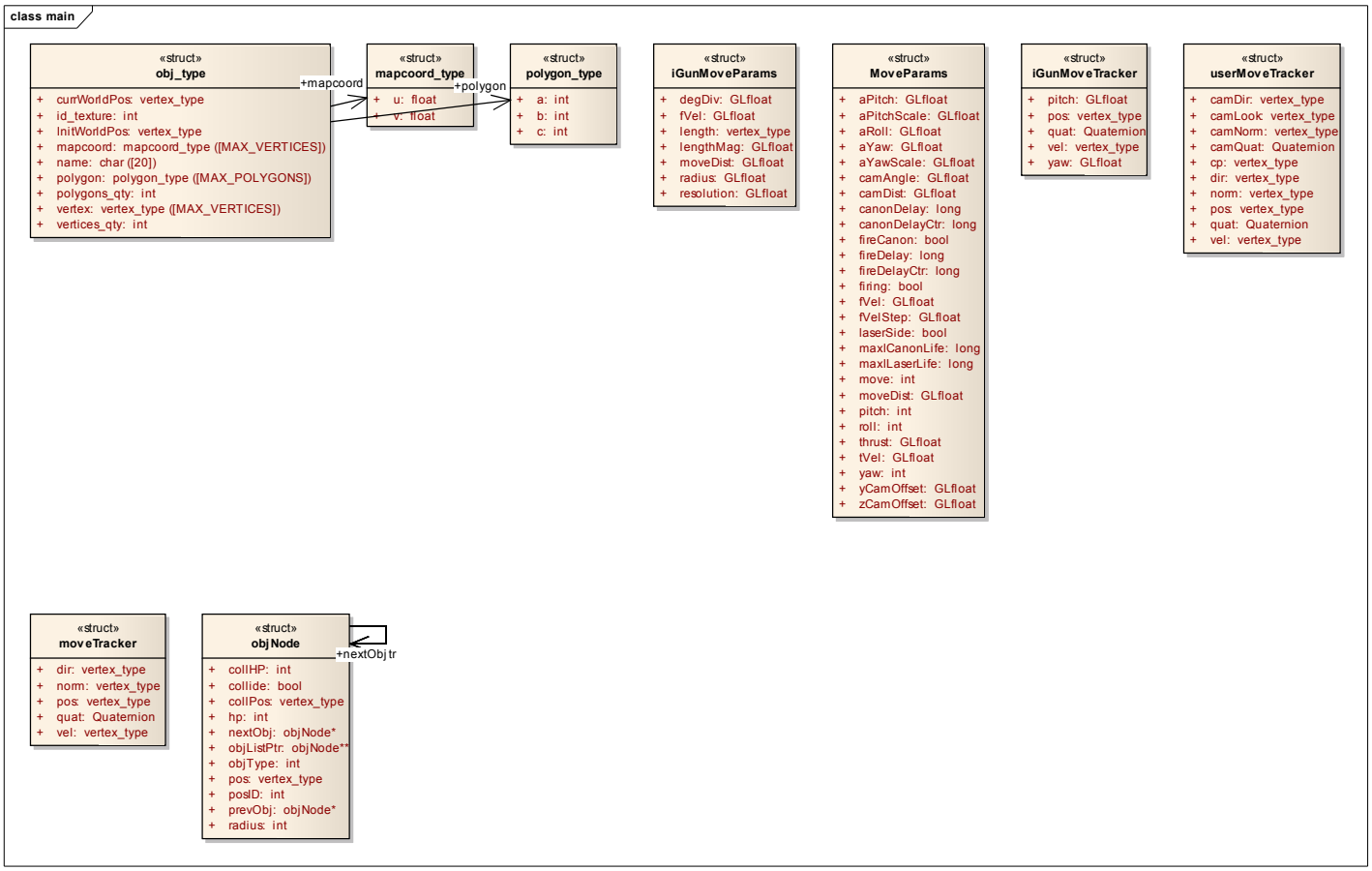

Figure 23 - Global Structures Class Outline

Starwarriors architecture contains __ main parts: The game loop engine, the particle engine, the graphic engine, the AI engine, the World engine, the input and sound engine and the object engine.

The game loop engine takes care of the entire game's initialization. It is responsible for calling all OpenGL initialization commands, setting up environment variables other engines need to use, fulfilling any requirements the included libraries will need. When everything is initialized, the game engine will enter the game main loop. In the main loop, each object will draw itself using the graphic engine and update their particular parameters according to the game timer or game logics. Special events such as keystrokes are monitored constantly in the main loop as well, updating any coordinating variables.

Initialization Phase

OpenGL initialization
SDL Initialization
Arena Initialization
UserFighter Initialization
Ai Fighter Initialization
Particle Engine Initialization

Runtime Phase

The game main loop is executed repeatedly to refresh both the screen and all relative variables. A display list is maintained by the game engine to manage all the objects that are needed to be drawn. Once the object is destroyed, the reference to this object will be deleted from the display list to prevent it from being drawn by graphic engine again.

Game Flow Diagram

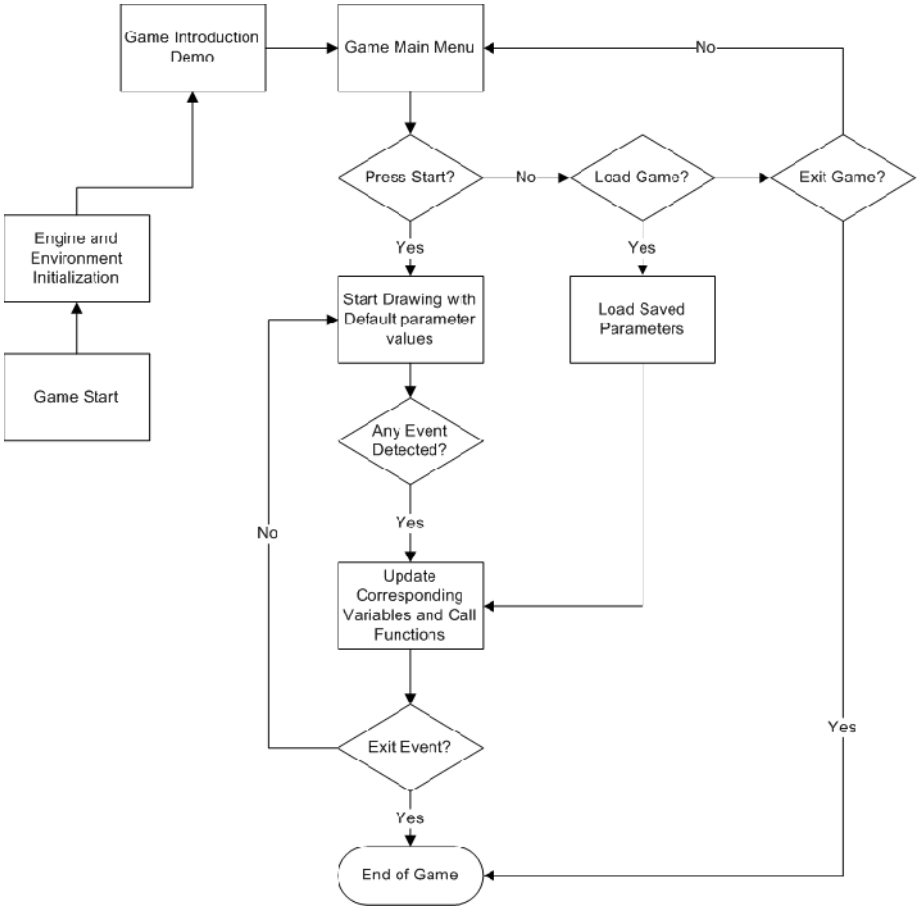

Figure 24 - Game Flow Diagram

Overall Game Engine Architecture

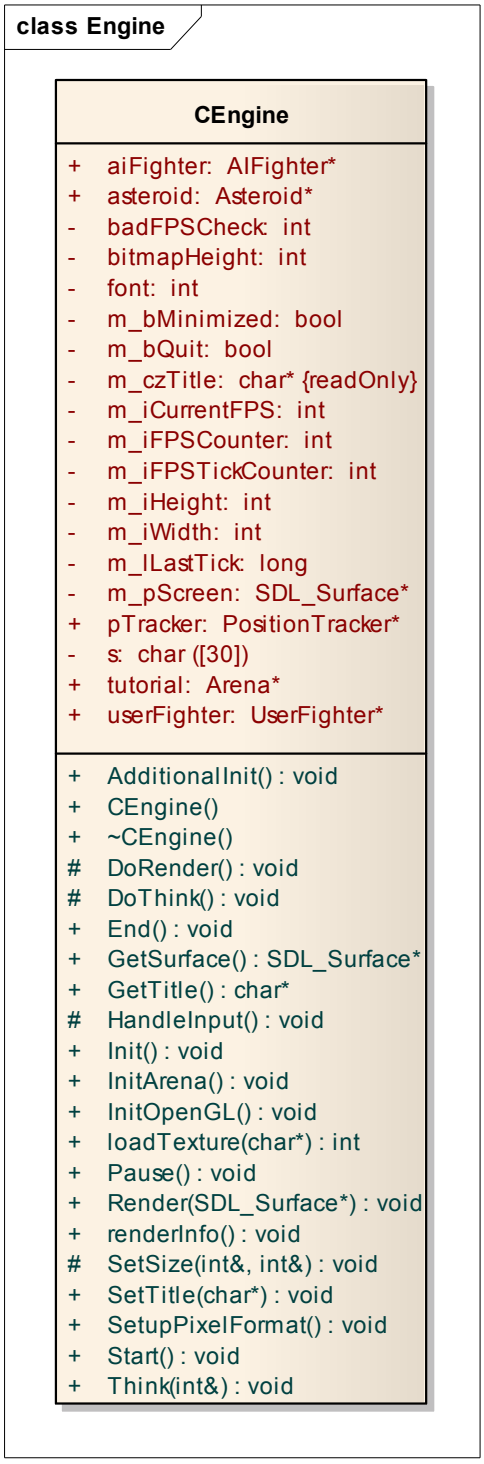

Figure 25 - Game Engine Class Outline

Detailed Module Structure

Quaternion/Math Model

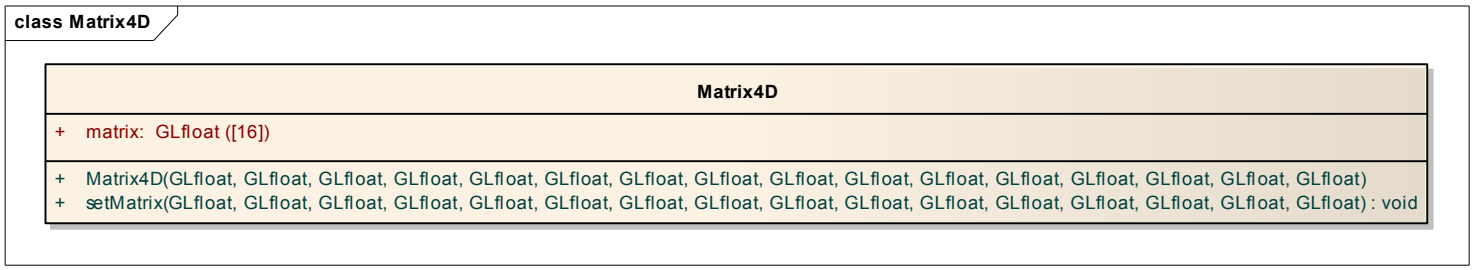

Figure 26 - 4 Dimensional Matrix Class Outline

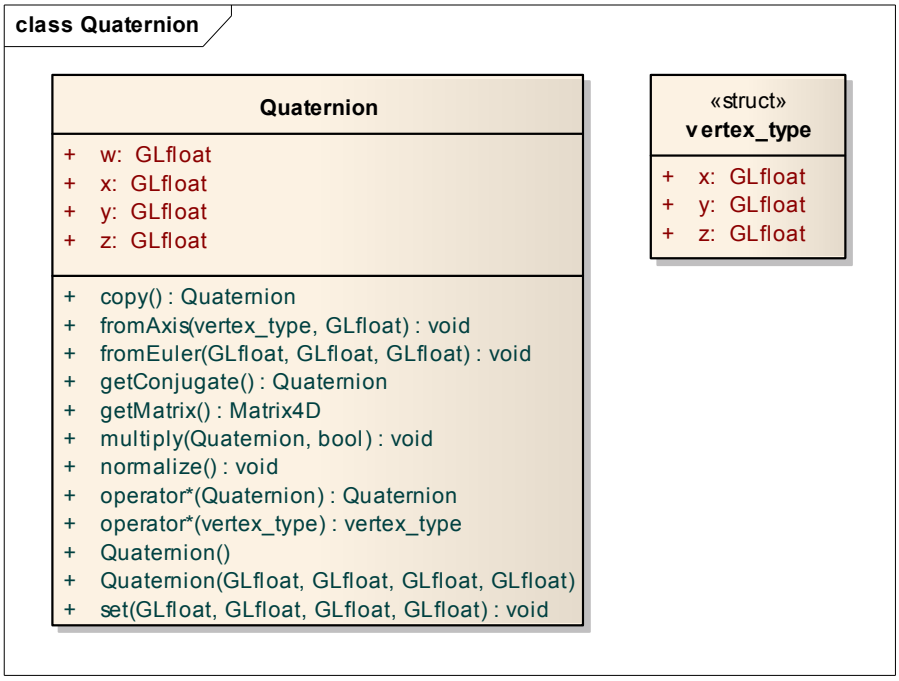

Figure 27 - Quaternion Class Outline

Arena

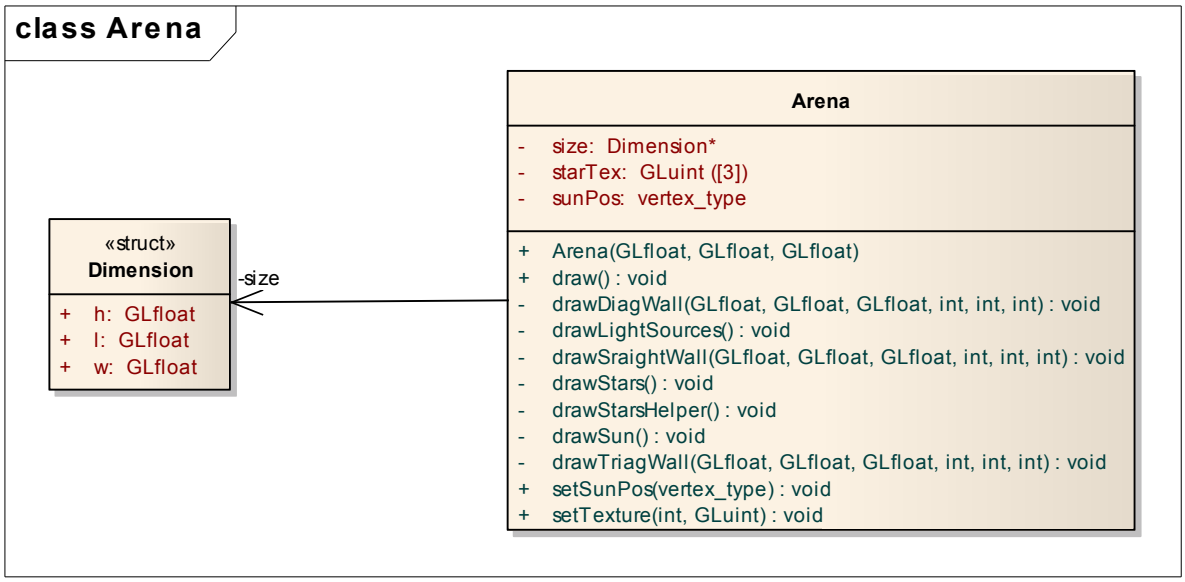

Figure 28 - Arena Class Outline

Collision Detection

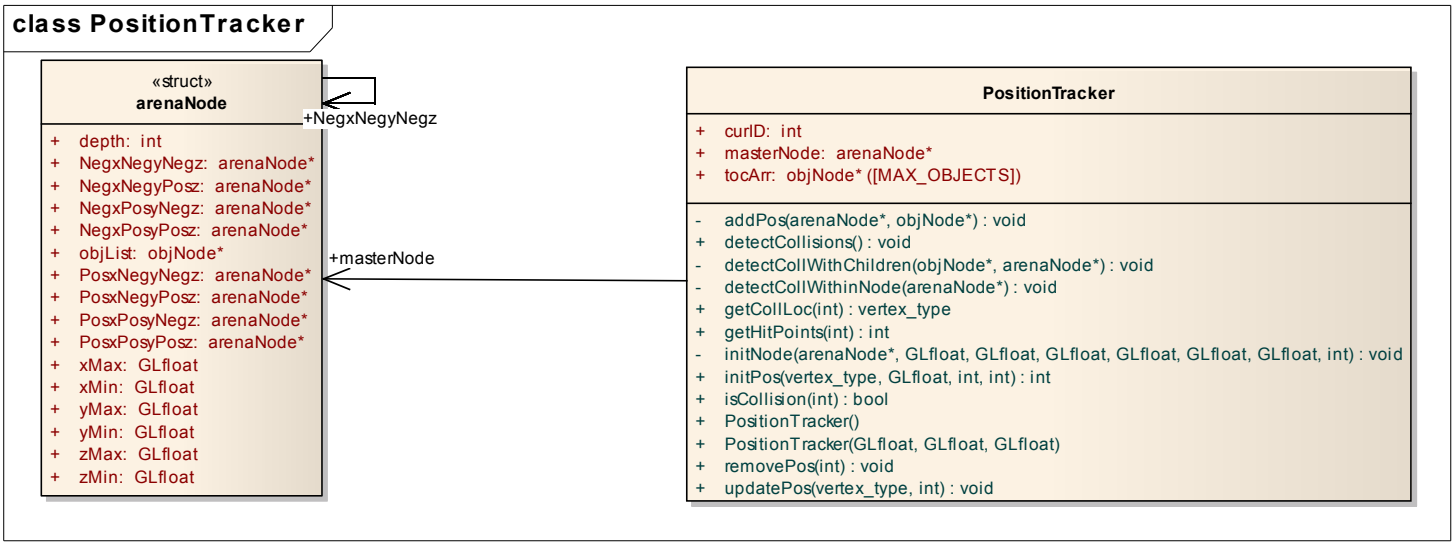

Figure 29 - Position Tracker Class Outline used for collision detection

Texture Loader

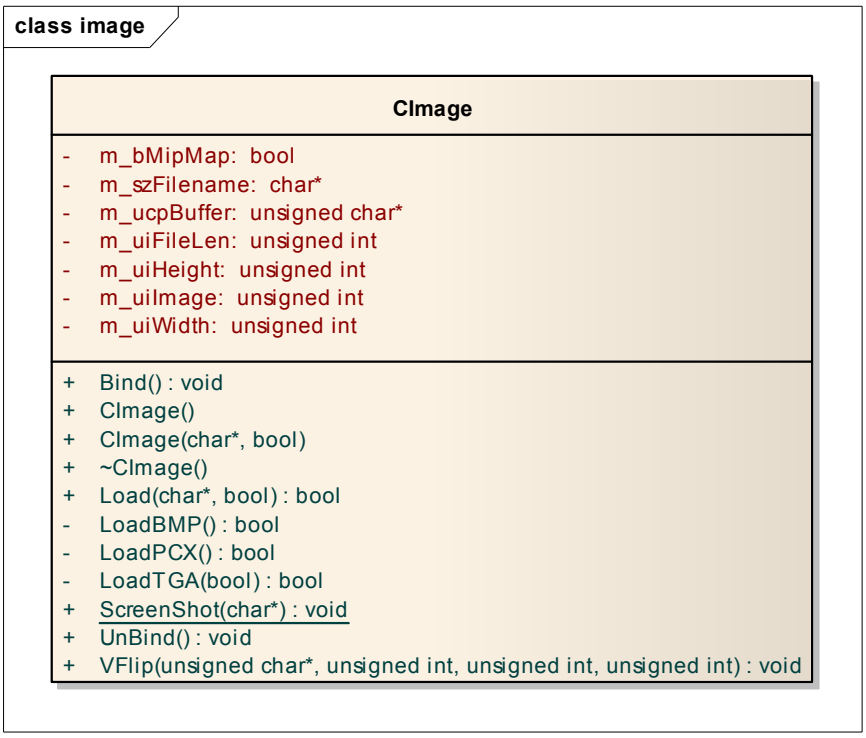

Figure 30 - Image Class Outline

Models

Asteroid

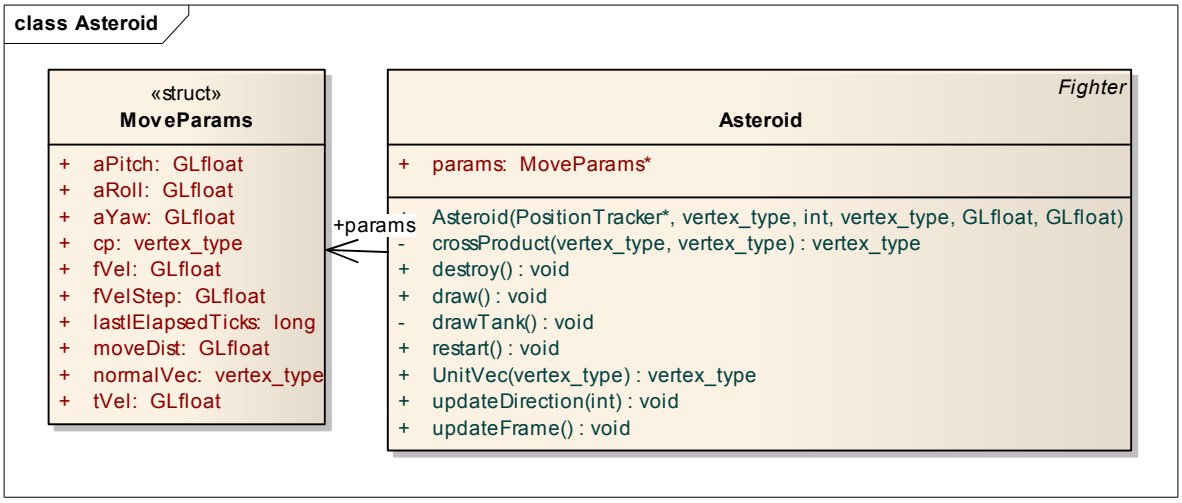

Figure 31 - Asteroid Class Outline

Fighter

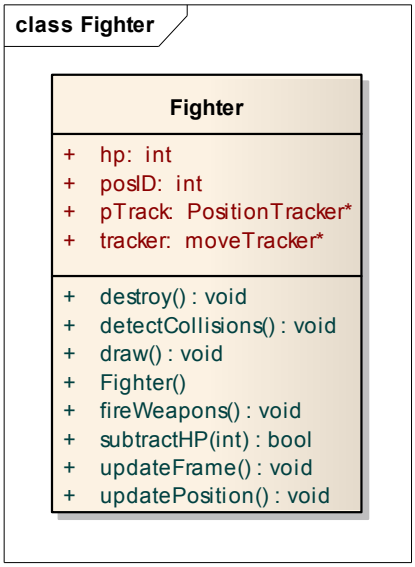

Figure 32 - Fighter Super Class Outline

Particle Engine

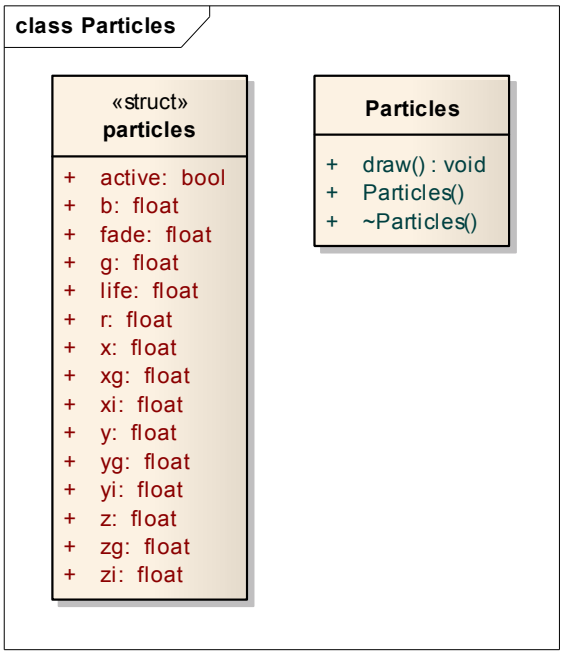

Figure 33 - Particle Engine Class Outline

Artificial Intelligence Fighter

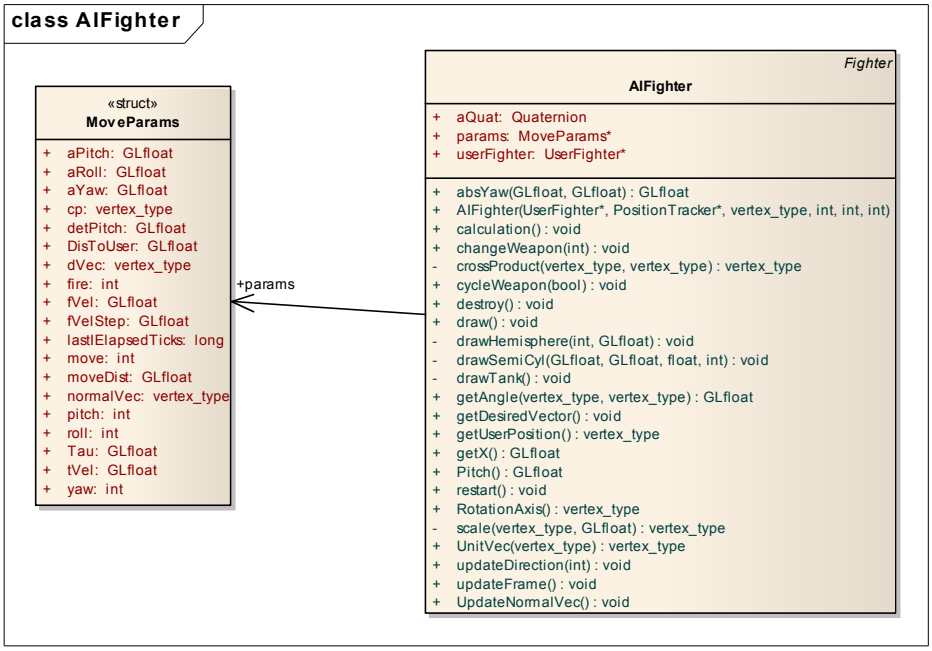

Figure 34 - Artificial Intelligence Fighter Class Outline

User Fighter

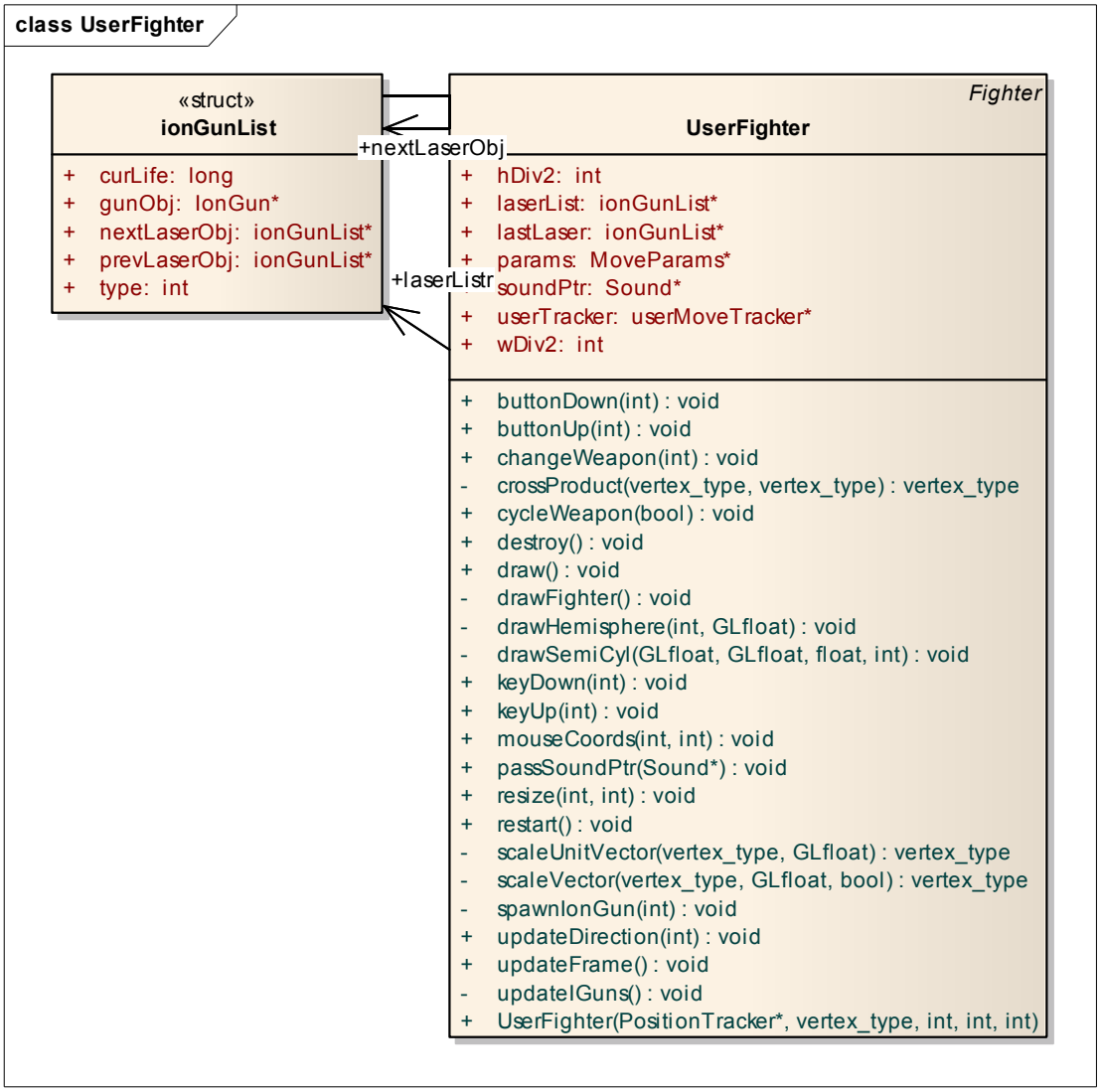

Figure 35 - User Fighter Class Outline

Ion Cannon

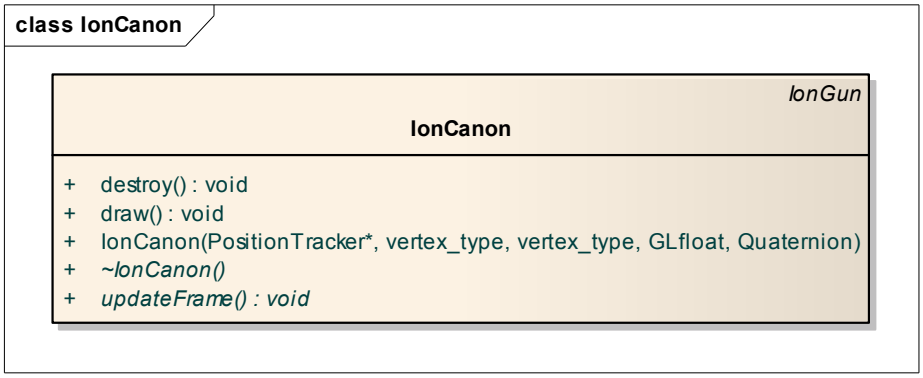

Figure 36 - Ion Cannon Class Outline

Ion Gun

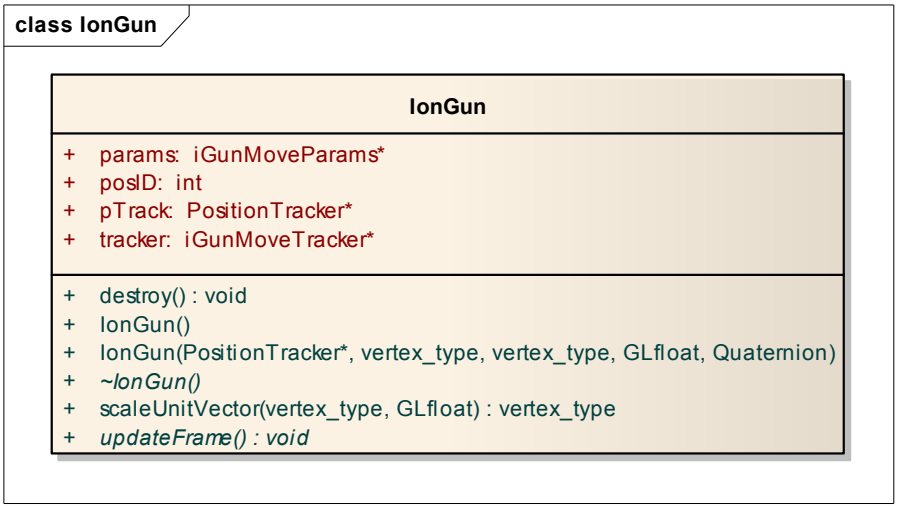

Figure 37 - Ion Gun Class Outline

Ion Laser

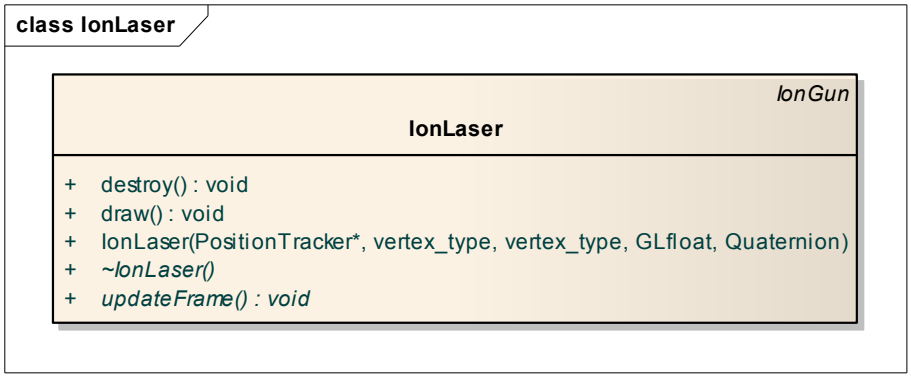

Figure 38 - Ion Laser Class Outline

User Interface

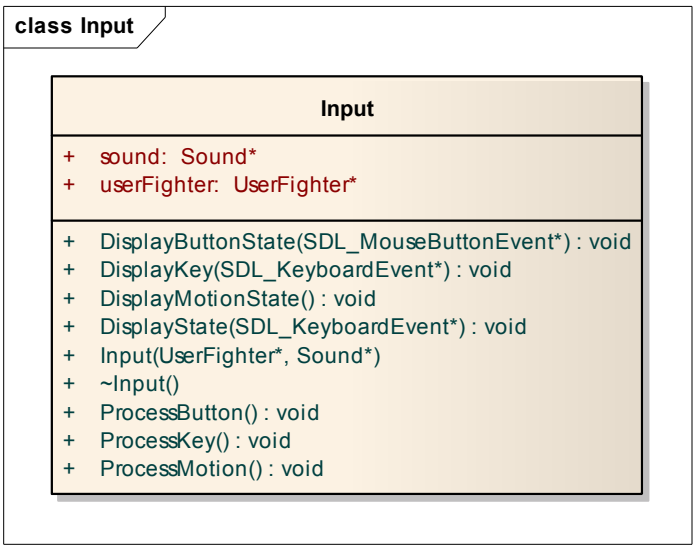

Figure 39 - Input Class Outline

Sound

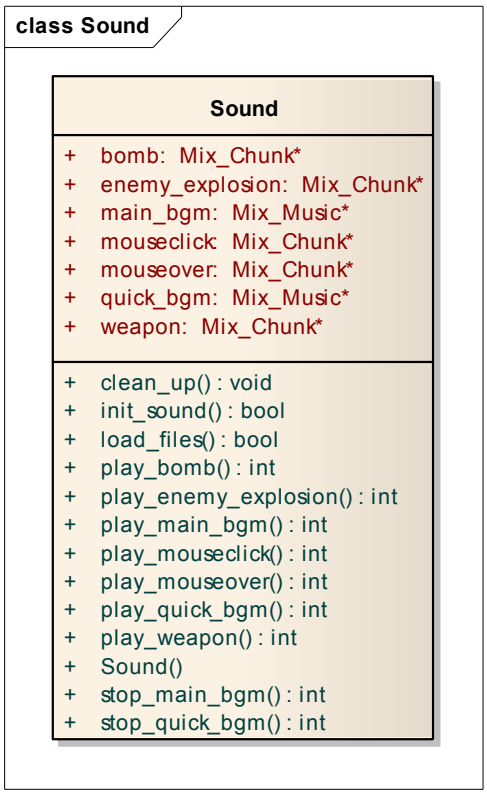

Figure 40 - Sound Class Outline

User Fighter Status and Game Metrics

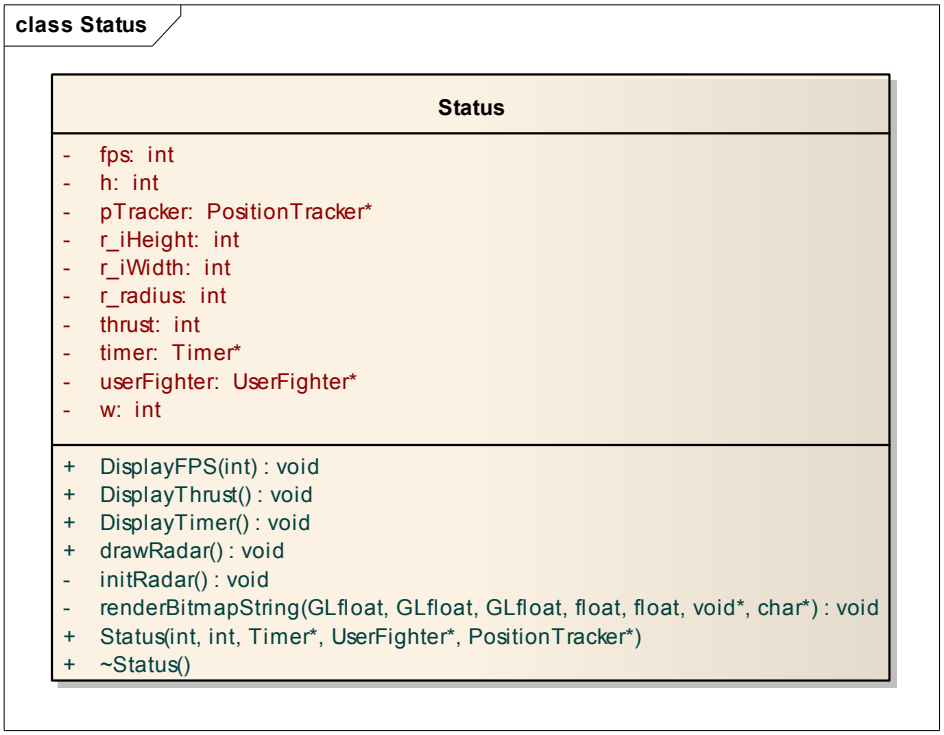

Figure 41 - User Fighter Status and Game Metrics Class Outline

Timer

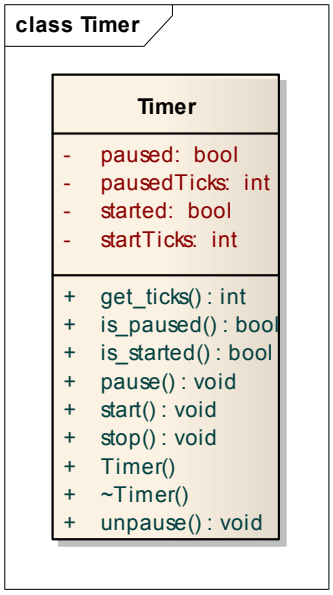

Figure 42- Timer Class Outline

Menu

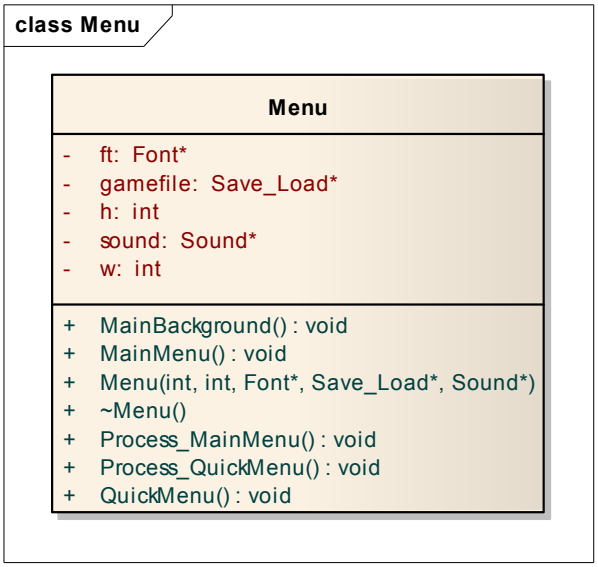

Figure 43 - Menu Class Outline

Save and Load

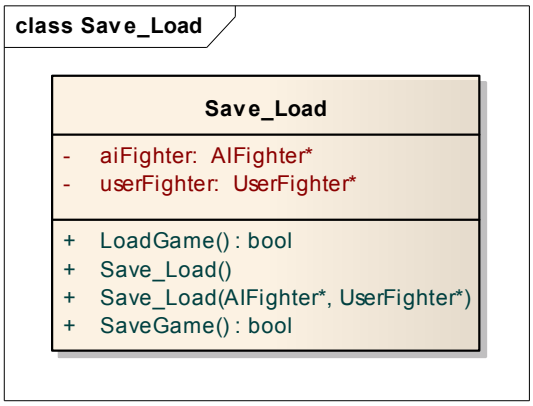

Figure 44 - Save and Load Class Outline

CONCLUSION

The tight schedule allotted for the game required a reduction in the scope of the requirements to exclude advanced features such as networked multiplayer over TCP/IP. However, many advanced features of the game play such as artificial intelligence, the third person camera and the engine were coded from scratch. Although this hard-coding does not offer as much flexibility as some libraries that have been in development for years, it is fine tuned for the Starwarriors game and offered a far greater learning experience than a richer feature set comprised of libraries ever could. Although there are bound to be bugs existing in the code, they are not readily noticeable and do not affect the very reasonable frame rate that can be achieved even on slow machines.

Because of the advanced nature of gaming calculations, there are undoubtedly bugs which still exist in the game. All critical bugs have been corrected however, leaving behind only minor glitches such as a laser sound not being played or skip in the frame rate periodically. Additional user acceptance and beta testing should be done before additional enhancements are made to the code, in order to reduce the complexity of debugging these enhancements.

The code operates at a frame rate of at least 60 FPS even on slow computers, which is sufficient for almost all monitors. Areas exist in the code where optimizations can be mode (such as the third person camera positioning calculations) but it was decided not to do so and rather focus on coding additional features. If this game is to be developed further, these code segments should be optimized in preparation for additional complex features which will decrease the frame rate.

APPENDIX A: Development Logs

The following is a list of individual logs for each member of the group. Coordination efforts such as Requirements and Architecture documentation, as well as task delegation can be viewed at:

http://code.google.com/p/eece478finalproject/
Username: starwarriors478
Password: eece478pass

Individual Development Log		**Lance Kersey**
Total Time:	104.00	
Date:	**Time Spent (Hours):**	**Work Description:**
15-Jan-08	1.00	Group Meeting
16-Jan-08	3.00	Researching 3D movement in space
16-Jan-08	10.00	Coding navigation backbone
22-Jan-08	1.00	Group Meeting
24-Jan-08	8.00	Researching Quaternions
25-Jan-08	10.00	Convert navigation to use quaternions
26-Jan-08	2.00	Researching third person camera positioning
29-Jan-08	1.00	Group Meeting
32-Jan-08	15.00	Programming third person camera
5-Feb-08	1.00	Group Meeting
12-Feb-08	1.00	Group Meeting
19-Feb-08	1.00	Group Meeting
26-Feb-08	1.00	Group Meeting
27-Feb-08	6.00	Researching collision detection
28-Feb-08	13.00	Programming collision detection
4-Mar-08	1.00	Group Meeting
11-Mar-08	1.00	Group Meeting
14-Mar-08	2.00	Researching view frustum culling techniques
18-Mar-08	1.00	Group Meeting
18-Mar-08	5.00	Programming culling
19-Mar-08	1.00	Researching third person camera visual movement feedback
19-Mar-08	2.00	Programming visual movement feedback
25-Mar-08	1.00	Group Meeting
6-Apr-08	6.00	Group Coding Session
12-Apr-08	5.00	Group Coding Session
13-Apr-08	5.00	Final Report

Individual Development Log		Adrian Yu
Total Time:	110.00	
Date:	**Time Spent (Hours):**	**Work Description:**
15-Jan-08	1.00	Group Meeting
22-Jan-08	1.00	Group Meeting
29-Jan-08	1.00	Group Meeting
3-Feb-08	2.00	Researching 3D Studio Max
5-Feb-08	1.00	Group Meeting
12-Feb-08	1.00	Group Meeting
15-Feb-08	2.00	Reading Tutorials on 3DS modelling
19-Feb-08	1.00	Group Meeting
26-Feb-08	1.00	Group Meeting
29-Feb-08	4.50	Modelling Ship in 3DS
1-Mar-08	5.00	Modelling Ship in 3DS
2-Mar-08	2.00	Reading Tutorials on 3DS loader
4-Mar-08	1.00	Group Meeting
7-Mar-08	4.00	Modelling Sun in 3DS
11-Mar-08	1.00	Group Meeting
12-Mar-08	3.00	Texturing Sun in 3DS
15-Mar-08	6.00	Modelling and Texturing Asteroids
16-Mar-08	4.00	Reading Tutorials on 3DS loader
17-Mar-08	3.00	Importing/Exporting 3DS models
18-Mar-08	1.00	Group Meeting
20-Mar-08	3.00	Researching 3DS exporting
25-Mar-08	1.00	Group Meeting
2-Apr-08	4.00	Researching 3DS exporting
3-Apr-08	5.00	Texture Mapping
5-Apr-08	8.00	Texture Mapping/Exporting 3DS Models
6-Apr-08	6.00	Group Coding Session
10-Apr-08	3.00	Researching 3D Studio Max Scene Animation
10-Apr-08	5.00	Creating 3DS animation video for opening scene
11-Apr-08	2.50	Created Final Report Template
12-Apr-08	5.00	Group Coding Session
12-Apr-08	6.00	Research Adobe Premier and created opening video scene
13-Apr-08	8.00	Final Report
14-Apr-08	8.00	Formatting/Integrating Final Report

Individual Development Log		Roger Yin
Total Time:	103.00	
Date:	**Time Spent (Hours):**	**Work Description:**
15-Jan-08	2.00	Group Meeting
22-Jan-08	1.00	Group Meeting
29-Jan-08	1.00	Group Meeting
3-Feb-08	2.00	Researching 3D Studio Max
5-Feb-08	1.00	Group Meeting
12-Feb-08	1.00	Group Meeting
15-Feb-08	2.00	Reading Tutorials on 3DS modelling
19-Feb-08	1.00	Group Meeting
26-Feb-08	1.00	Group Meeting
29-Feb-08	4.50	Modelling Ship in 3DS
1-Mar-08	5.00	Modelling Ship in 3DS
2-Mar-08	2.00	Reading Tutorials on 3DS loader
4-Mar-08	1.00	Group Meeting
7-Mar-08	4.00	Modelling Sun in 3DS
11-Mar-08	1.00	Group Meeting
12-Mar-08	3.00	Texturing Sun in 3DS
15-Mar-08	6.00	Modelling and Texturing Asteroids
16-Mar-08	4.00	Reading Tutorials on 3DS loader
17-Mar-08	3.00	Importing/Exporting 3DS models
18-Mar-08	1.00	Group Meeting
20-Mar-08	3.00	Researching 3DS exporting
25-Mar-08	1.00	Group Meeting
2-Apr-08	4.00	Researching 3DS exporting
3-Apr-08	5.00	Texture Mapping
5-Apr-08	8.00	Texture Mapping/Exporting 3DS Models
6-Apr-08	6.00	Group Coding Session
10-Apr-08	3.00	Researching 3D Studio Max Scene Animation
10-Apr-08	5.00	Creating 3DS animation video for opening scene
11-Apr-08	2.50	Created Final Report Template
12-Apr-08	5.00	Group Coding Session
12-Apr-08	6.00	Research Adobe Premier and created opening video scene
13-Apr-08	8.00	Final Report

Individual Development Log		Chris Lee (Sangkyu Lee)
Total Time:	105.00	
Date:	**Time Spent (Hours):**	**Work Description:**
15-Jan-08	1.00	Group Meeting
22-Jan-08	1.00	Group Meeting
29-Jan-08	1.00	Group Meeting
3-Feb-08	1.50	Studying with SDL Tutorial
5-Feb-08	1.00	Group Meeting
12-Feb-08	1.00	Group Meeting
15-Feb-08	2.00	Studying with SDL Tutorial
19-Feb-08	1.00	Group Meeting
26-Feb-08	1.00	Group Meeting
29-Feb-08	5.00	Start building the "Input" Class
1-Mar-08	5.00	Working on the "Input" class
2-Mar-08	2.50	Completed the "Timer" class
4-Mar-08	1.00	Group Meeting
5-Mar-08	5.00	Test the "Input" Class with simple moving square program and debugging
11-Mar-08	1.00	Group Meeting
13-Mar-08	2.50	Researching on the SDL extension libraries
14-Mar-08	5.00	Complete the "Input" Class
15-Mar-08	2.00	Reading the tutorials on the SDL sound extension libraries
16-Mar-08	3.00	Start building the "Sound" class
17-Mar-08	3.00	Test the "Sound" class and search for more sound effects
18-Mar-08	1.00	Group Meeting
20-Mar-08	3.00	Searching for the background music and working on the "Sound" class
25-Mar-08	1.00	Group Meeting
2-Apr-08	5.00	Completed the "Sound" class and start the "Font" class
3-Apr-08	5.50	Completed the "Font" class and start "Menu" class using "Font" class
5-Apr-08	7.00	"Menu" class completed
6-Apr-08	6.50	Group Coding Session
10-Apr-08	3.00	Start making "Status" class using "Font" class
10-Apr-08	4.00	Completed "Status" class
11-Apr-08	2.50	Added more feature for "Sound" class and "Font" class
12-Apr-08	5.00	Group Coding Session
12-Apr-08	7.00	Fixing bugs for the "Menu" class and work on the "Save_Load" class
13-Apr-08	9.00	Final Report

Individual Development Log		Hyun Woo Choi
Total Time:	**104.00**	
Date:	**Time Spent (Hours):**	**Work Description:**
15-Jan-08	1.00	Group Meeting
22-Jan-08	1.00	Group Meeting
29-Jan-08	1.00	Group Meeting
4-Feb-08	1.00	Research
5-Feb-08	1.00	Group Meeting
12-Feb-08	1.00	Group Meeting
19-Feb-08	1.00	Group Meeting
25-Feb-08	3.00	Software Requirement Specification Document
26-Feb-08	1.00	Group Meeting
28-Feb-08	3.00	AI tracking
29-Feb-08	5.00	AI orientation research
2-Mar-08	3.00	Testing
4-Mar-08	1.00	Group Meeting
7-Mar-08	2.00	Research on AI
11-Mar-08	1.00	Group Meeting
12-Mar-08	2.00	Research on AI
15-Mar-08	2.00	AI Math Research
18-Mar-08	1.00	Group Meeting
20-Mar-08	6.00	AI Math Research
21-Mar-08	5.00	AI Math Research
22-Mar-08	4.00	AI Math
25-Mar-08	1.00	Group Meeting
1-Apr-08	4.00	AI Math
2-Apr-08	2.00	Research on Sound effect
3-Apr-08	3.50	Research on Sound Effect Files
4-Apr-08	4.00	AI Math
5-Apr-08	4.00	AI Implementation
6-Apr-08	6.00	Group Coding Session
7-Apr-08	3.50	AI Implementation
8-Apr-08	6.00	Debug
9-Apr-08	5.00	Debug
10-Apr-08	2.00	Modification on AI implementation
11-Apr-08	2.00	Asteroid Implementation
12-Apr-08	6.00	Group Coding Session/ Asteroid Implementation
13-Apr-08	9.00	Final Report

REFERENCES

1. OpenGL: Tutorials: Using Quaternions to Represent Rotation <http://gpwiki.org/index.php/OpenGL:Tutorials:Using_Quaternions_to_represent_rotation >
2. Dopertchouk, Oleg. *Simple Bounding-Sphere Collision Detection* <http://www.gamedev.net/reference/articles/article1234.asp>
3. Angel, E (2006). *Interactive Computer Graphics: A Top-Down Approach Using OpenGL.* (4th Ed). United States of America: Pearson Education, Inc.
4. Astle, D & Hawkins, K. (2004). *Beginning OpenGL Game Programming*. United States of America: Premier Press.
5. Cogswell, J. (2003). *C++ All-In-One Desk Reference for Dummies.* Indianapolis, Indiana: Wiley Publishing, Inc.
6. Dawson, M (2004). *Beginning C++ Game Programming.* United States of America: Thomson Course Technology PTR.
7. Molofee, Jeff. Neon Helium. April 2008. <http://nehe.gamedev.net>
8. Pipho, E. (2003). *Focus on 3D Models.* United States of America: Premier Press.
9. Gamedev.net. *Quaternion Powers* (2008) <http://www.gamedev.net/reference/articles/article1095.asp>
10. *Maths – vectors, Euclidean Space - building a 3D world* (2008). <http://www.euclideanspace.com/maths/algebra/vectors/index.htm>
11. *The cosine Law, COOL School- online content development* (2008). <http://www.coolschool.ca/lor/AMA10/unit7/U07L04.htm>
12. Lighthouse 3D - GLUT Tutorial : <http://www.lighthouse3d.com/opengl/glut/>
13. Simple Direcmedia Layer: <http://www.libsdl.org/>

14. Lazy Foo' Productions (SDL Tutorial): <http://lazyfoo.net/SDL_tutorials/>

15. Partner In Rhyme (Sound): <http://www.partnersinrhyme.com/soundfx/warsounds.shtml>

16. ACOUSTICA (Sound): <http://www.acoustica.com/sounds.htm>

17. Horton, I. (2005). *Ivor Horton's Beginning Visual C++ 2005* (from wrox)